USENIX Association

Proceedings of the The Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS)

June 17-21, 1996

Toronto, Canada

Table of Contents

The Second USENIX Conference on Object-Oriented Technologies and Systems (COOTS)

June 17-21, 1996
Toronto, Ontario, Canada

Wednesday, June 19

Opening Remarks
Douglas C. Schmidt, Washington University

Keynote Address:
Experiences on the Road to Object Utopia: An Industrial Research and Development Perspective
Dave Thomas, Object Technology International

C++
Session Chair: Daniel Edelson, IA Corporation

Compiler Optimization of C++ Virtual Function Calls 3
Sara Porat, David Bernstein, Yaroslav Fedorov, Joseph Rodrigue, and Eran Yahav, IBM Haifa Research Laboratory

Composing Special Memory Allocators in C++ 15
Keith Loepere, Open Group Research Institute

Building Independent Black Box Components in C++ 25
Mark Addesso, Software AG of North America

CORBA and Distributed Objects
Session Chair: Steve Vinoski, Hewlett-Packard

Interlanguage Object Sharing with SOM 35
Jennifer Hamilton, IBM Toronto Laboratory

Extending a Traditional OS Using Object-Oriented Techniques 53
Jose Bernabeu-Auban, Vlada Matena, and Yousef Khalidi, Sun Microsystems Laboratories

Object Caching in a CORBA Compliant System 65
R. Kordale and M. Ahamad, Georgia Tech; M. Devarakonda, IBM T. J. Watson Research

Asynchronous Notifications Among Distributed Objects 83
Yeturu Aahlad, Bruce E. Martin, Mod Marathe, and Chung Lee, SunSoft, Inc.

Tools
Session Chair: Doug Lea, SUNY Oswego

Preliminary Design of ADL/C++ – A Specification Language for C++ 97
Sreenivasa Rao Viswanadha, SUNY Albany; Sriram Sankar, Sun Microsystems Laboratories

Software Composition with Extended Entity-Relationship Diagrams 113
Pornsiri Muenchaisri and Toshimi Minoura, Oregon State University

A Measure of Testing Effort 129
John McGregor and Satyaprasad Srinivas, Clemson University

Thursday, June 20

Patterns
Session Chair: Doug Schmidt, Washington University

Design Patterns for Dealing with Dual Inheritance Hierarchies in C++ 143
Robert Martin, Object Mentor

The Object Group Design Pattern 155
Silvano Maffeis, Olsen & Associates, Zurich Cornell University, Switzerland

Pattern Languages for Handing C++ Resources in an Exception-Safe Way 165
Harald M. Müeller, Siemens AG, Austria

Object-Oriented Frameworks and Components
Session Chair: Jim Waldo, Sun Microsystems Labs and JavaSoft

The Any Framework: A Pragmatic Approach to Flexibility 179
Kai-Uwe Mätzel and Walter Bischofberger, UBILAB

Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging 191
Irfan Pyarali, Timothy H. Harrison, and Douglas C. Schmidt, Washington University

Class Relationships and User Extensibility in Solid Geometric Modeling 209
James R. Miller, University of Kansas

Distribution Languages
Session Chair: Vince Russo, Purdue University

A Distributed Object Model for the Java™ System 219
Ann Wollrath, Roger Riggs, and Jim Waldo, JavaSoft

Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems 233
Eshrat Arjomandi, York University, Toronto; William G. O'Farrell and Gregory V. Wilson, IBM Canada Ltd.

Pickling State in the Java™ System 241
Roger Riggs, Jim Waldo, Ann Wollrath and Krishna Bharat, JavaSoft

Highly Concurrent Distributed Knowledge Objects 251
K.L. Clark and T.I. Wang, Imperial College, London

Conference Organizers

Preface

Welcome to the 2nd USENIX Conference on Object-Oriented Technologies and Systems (COOTS). Like the USENIX C++ Conferences from which it evolved, COOTS is dedicated to showcasing advanced R&D work on object-oriented technologies and software systems. This year's technical program presents recent results from research on, and development of, object-oriented frameworks and components, design patterns, CORBA, C++, and Java. Particular emphasis this year is on software architectures, tools, and programming languages that support distributed object computing.

The 20 papers you'll hear about over the next two days were selected from a total of 50 submitted to the conference. I'm grateful for the help of the program committee in reviewing the submissions in a thoughtful and timely manner. In particular, Doug Lea, Vince Russo, Steve Vinoski, and Jim Waldo did a stellar job helping to assemble the final technical program. Doug Lea also deserves the credit for arranging our strong tutorial program.

This year's COOTS has several noteworthy aspects. First, after a one year hiatus, we're rekindling a tradition from earlier USENIX C++ conferences: the Advanced Topics Workshop. This one-day, post-conference workshop will focus on the development of methods, tools, and services supporting distributed object computing on the Internet. Topics covered at this year's workshop include the performance, scalability, reliability, management and security issues surrounding JAVA and content-oriented languages for the WWW, as well as CORBA and Network OLE in large-scale distributed applications. I'd like to thank David Cohn for organizing and chairing the Advanced Topics Workshop.

Second, the final session on Thursday will employ the "writer's workshop" format to review a selected set of technical papers. Writer's workshops are a form of "active participation" commonly used to review papers in the Patterns Community. However, COOTS is one of the first conferences to employ this technique to improve the quality of technical papers. These papers will be published in revised and expanded form in a special issue of the USENIX *Computing Systems* journal, so I strongly encourage you to contribute your insights and experience to authors in the writer's workshop.

Finally, I'd like to thank Ellie, Judy, Carolyn, Toni, and Dan at USENIX. They provided me with invaluable help planning, organizing, and promoting COOTS. Not only did they make the whole process run smoothly, they also made it educational and enjoyable. I hope that this week in Toronto, you'll find the fruits of our efforts educational and enjoyable, as well.

Douglas C. Schmidt
Washington University, St. Louis

Compiler Optimization of C++ Virtual Function Calls

Sara Porat
David Bernstein
Yaroslav Fedorov
Joseph Rodrigue
Eran Yahav
VNET: porat at haifasc3
E-mail: porat@haifasc3.vnet.ibm.com
Tel: (972) 4 8296309
FAX: (972) 4 8296114
IBM Haifa Research Laboratory
Matam, Haifa 31905, Israel

Abstract

We describe two generic optimization techniques to improve run-time performance of C++ virtual function calls: *type specification* and *type prediction*. Both involve program analysis that results in a set of call sites to be optimized, and code transformations that replace the original dispatching mechanism in these sites by more efficient call expressions. We implement two special cases. The first is a type-specification optimization, called *unique name*, that requires static global view of the whole program in order to substitute indirect virtual function call sites by direct calls. The other is a type-prediction kind of optimization, referred to as *single type prediction*, that uses type-profiling information to replace virtual function calls by conditional expressions which involve direct function calls.

These optimizations were implemented in IBM's compiler, the C Set ++ for AIX/6000, and evaluated on a set of C++ standard benchmarks. We received encouraging run-time results with improvements in order of 20% that demonstrate the vitality of these techniques.

1 Introduction

One of the most powerful concepts in C++ programming is *polymorphism* as provided by virtual functions. This is a powerful abstraction mechanism, but it carries on a significant run-time penalty. A virtual function call is usually implemented as an indirect function call through a table of functions generated by the compiler, so called Virtual Function Table or VFT. This indeed ensures reasonable efficiency for such calls, but it has by no doubt certain run-time overhead as compared to direct function calls. Moreover, and possibly even more important with respect to performance criteria, is the fact that a compiler cannot apply inline substitution to unqualified virtual function calls. Such inline substitution could have totally eliminate the calling overhead, and enable the compiler to further apply standard optimizations on inlined bodies of virtual functions.

In this study, we analyze and develop techniques to optimize virtual function calls in C++. We were motivated by other studies ([6]) that convince us that performance is most critical for C++, and that efficiency problems can become a key inhibitor to widespread acceptance of this language.

The subject of our work is replacing indirect virtual function call expressions in C++ by more effective expressions that include direct calls to virtual functions. In some cases, a static analysis of the whole program [4] can determine for a particular virtual function call site that there is a *unique* implementation of this function that can possibly be called at that point. In such case, the virtual function call can be replaced by a direct call. Even when the particular virtual function call is not always resolved to a single implementation, we may explore type-feedback information [10] that indicates that there is a particular implementation that is resolved most of the time. Using this information, an indirect virtual function call can be replaced by a *conditional* (type-predicted) expression, so that the likely invoked function is called directly and others are invoked through the regular indirect mechanism.

The topic of virtual function call optimization has been of great interest in the last years. Some papers ([8], [11]) just report dynamic program characteristics of C++ programs and show that virtual calls are very common in C++ programming style. These studies are not proposing solutions, but provide the basic motivation for subsequent research works. Others, like [7], propose techniques for optimizing virtual function calls, e.g. by transforming them to conditional expressions, and analyze the effects of such optimizations, but do not provide concrete implementations. An interesting comparison of optimization techniques in the context of SELF can be found in [1]. In contrast, we describe *compiler-based implementations* that *find* opportunities for optimizations, and consequently *apply* code transformations.

Many papers are closely related to our work and introduce similar implementations to optimize virtual function calls. We cite only a subset of all these works, and primarily reveal the differences between them and ours. Other interesting studies can be found in those papers that we do reference.

Algorithms that determine for virtual function call sites whether or not they are resolved to a unique implementation can be found in [3], [4] and [12]. These papers focus on whole-program analysis of C++ code but none of them involve compiler implementations to generate optimized virtual calls.

Run-time type feedback is used in [10] and [9] to optimize dynamically-dispatched calls in SELF and Cecil. This technique is also examined in [9] in the context of C++, but it is limited to the stage of code analysis only, i.e. no transformations are applied to the code.

A very recent paper by Aigner and Holzle ([2]) (that was published after a draft of our paper was already submitted for publication) seems to be the most related work to that presented here. The main distinction between their approach and ours is that they use a source-to source process to compile a *baseline all-in-one* C++ source into C++ optimized code, whereas we change an *existing production compiler* that compiles original source code and generates optimized code. Our solution is much more in the spirit of other compiler optimization techniques, and seems to be much more efficient than that proposed in [2]. To conclude, our work pioneers the use of an existing C++ compiler to close the whole loop of optimizing C++ virtual function calls: performing program analysis, evaluating the results of the analysis, feeding back the results to the compiler, and generating optimized virtual function calls given this information.

A different approach for optimizing virtual function calls is introduced through *subtype cloning*. This technique removes dynamic dispatches resulting from parametric polymorphism through code replication. A recent work [13] presents an efficient cloning algorithm, and a comprehensive summary of related work.

Our development environment consists of IBM's production compiler, the C Set ++ for AIX/6000 ([14]). Here the compilation process consists of several code processing phases: front-end processing, subsequent function inlining processing, the invocation of which is controlled by the user, and which attempts to inline functions that are marked as inline, and finally back-end processing. Since the inliner is invoked right after the front-end, the idea is to let the front-end detect and exploit the possibilities for virtual function call conversions, so as to allow subsequent inlining of transformed virtual function calls. In addition to that, the back-end is not capable of naturally detecting virtual function calls and optimizing them, since its intermediate language lacks the high-level concepts of virtual functions and VFTs. Thus, our goal is to implement a variation of the C++ compiler front-end that will automatically replace virtual function calls when possible and when such replacement will reduce the path-length. We strongly believe that such optimization can play a major role in optimizing C++ programs, since it reduces the calling overhead and allows further back-end optimization as a result of inlining.

We apply our modified C++ compiler to several C++ programs, and the results so far are very encouraging. Of special interest are those programs that make extensive use of virtual functions. One of these is Tal-Dict, a performance test program of the Taligent dictionary class, where our optimization succeeds to improve the execution time by 20%.

The remainder of this paper is organized as follows. In the following section we describe our optimization techniques in general terms of type specification and type prediction. In Section 3 we illustrate our optimizations with a small example program. In Section 4 we give more details on our implementation work with the unique-name optimization and the single-type-prediction optimization. Next, in Section 5 we evaluate our techniques and present the benchmarks that we used to experiment our optimizations. Finally, in Section 6 we discuss our conclusions and plans for future work.

2 Optimization via Analysis and Transformation

In this section we will clarify the key-concepts in our optimizations, without getting into specific implementation details.

Our optimizations comprise two phases, *analysis* and *transformation*. In the first, we collect information about the program to result in a set of *candidate* unqualified virtual function call (VFC) sites. Each such candidate identifies an exact source location (file name, line and column) of a particular call expression that invokes a virtual function through the VFT mechanism. A candidate VFC also introduces a set of *specified* or *predicted* class names. In the second phase, we use the candidate set to apply transformations that try to improve performance. Each candidate VFC is optimized using its corresponding set of class names.

2.1 Type Specification and Type Prediction

Type-specification optimization is aimed at transforming a set of candidate VFC sites. Each candidate VFC is associated with a list of *specified* class names. Each of these class names defines a possible target function implementation that may be invoked in this call expression. The whole list associated with a particular candidate VFC represents *all* possible target functions to which this call can be resolved. In the transformation phase we look up the list of n target functions corresponding to each candidate call site, and replace the indirect call with an n-way branch expression to direct calls to the target functions.

In the analysis phase of type-specification optimization we need to traverse the complete class hierarchy. Moreover, determining for a particular VFC site the precise set of target functions is NP-Hard ([12]). Each approximation results in pessimistic sets, i.e. some target function implementations that are listed for a particular VFC site may in fact never be invoked. Another important consideration is the cost of an n-way branch, for $n > 1$. In most architectures such branch is implemented through a complex conditional expression that depends on nested comparisons, each of which compares the type of the object, against which the call is currently invoked, with some specified type. Usually type-specification transformation should only be performed if n is smaller than some threshold value, depending on the cost of conditional branches on the target machine and the effectiveness of the optimizer. These considerations may limit the size of the candidate set in type-specification optimization, and hence its applicability for some programs.

Type-prediction optimization is aimed at transforming a larger set of candidate VFC sites than in the case of type specification. Here we identify for each VFC site a set of m *predicted* class names. Each predicted class name defines a particular predicted function implementation, that may be invoked in this call expression, and probably more often than other implementations. In the transformation phase we replace the unqualified call with an $m + 1$-way branch, so that the predicted functions are called directly, and all the rest are invoked through the regular dispatching mechanism.

In this research, we present only two special cases of these generic optimizations. *Unique name* (UN) is a special case of type specification, where the transformation is done only if the set of possible target functions is a singleton, i.e. $n = 1$. In *single type prediction* (STP) we identify, for each candidate VFC site, a unique predicted target, i.e. $m = 1$. Our chosen predicted function is the one that is called most of the time.

2.2 Unique Name

The analysis phase for this optimization consists of finding VFC sites for which only one destination exists. For each UN VFC candidate, a unique class name is identified.

There are many methods suggested in the literature to find a UN candidate set, varying from simple algorithms ([3] [7]) up to approximations using class-hierarchy pruning ([4]) and complex data flow analysis ([12]).

We implement a simple algorithm that explores information on VFC sites and knowledge on the complete class-hierarchy. For each VFC, we find all of the *possible methods* that can be called at that point. The basic algorithm that we used is referred in [3] as *simple call graph construction*. For more implementation details see Section 4. The set of UN candidates can be obtained using any program-database or compiler based tool. In any case, the analysis phase requires a global view of the whole program.

In the transformation phase, we chose to visit all the candidate VFC sites found, and replace the indirect function call expression by a direct call expression to the corresponding destination.

An alternative way to explore the fact that some virtual function has a unique implementation over some class hierarchy is by *nullifying its virtuality*, i.e. modify the characterization of this function as a virtual function, and let it be non-virtual. This approach can be easily implemented within a compiler front-end. From a performance perspective, the affects of such a transformation are similar to those gained by transforming UN candidates and get them be direct calls. On the other hand, removing the virtual characterization has the advantage of reducing code size by removing entries in VFTs. As far as we know, this technique hasn't been yet implemented and evaluated.

2.3 Single Type Prediction

STP optimization is a special case of type-prediction optimization in which we identify for each VFC site a single target function that is called most of the time. In this case, we replace the indirect call by a conditional expression so that the likely invoked function is called directly and others through the regular indirect call mechanism.

We chose to find STP candidates using off-line profiling information, as it is done in [2]. (In contrast, the SELF system ([10]) uses on-line profiling information.) We implement a tool that provides information about how often different target functions are invoked when the program is running. Our tool shows for each VFC site how many times each possible method was invoked over a series of executions of the program. Every VFC that is invoked in these executions serves as an STP candidate, and its associated predicted class name is the one that defines the target function that was invoked more (or at least not less) than each other target function.

Note that profiling analysis requires running the program on a series of inputs. For those applications where profiling is too expensive, STP candidates may be somehow provided by the programmer.

To implement our profiling tool, we use a mechanism similar to the Test Coverage Tool in C Set ++ for AIX/6000. We implement a variation of the compiler that instruments every compilation unit of the program to facilitate collection of type-profiling information about VFC sites.

In the STP transformation phase we visit all candidate VFC sites, and replace the indirect function call expression <indirect-call> by a conditional expression of the form <exp> ? <direct-call> : <indirect-call>, where <exp> is evaluated to TRUE if we reach the VFC site with a *predicted-type* object. When reaching the site with an object of another type, the original indirect call is taken. The direct call in the expression invokes the predicted target function. In what follows, we refer to <exp> as the *type-info* expression.

When we have a *hit* (an object of the predicted type reaches the call site), we only pay the overhead of evaluating the type-info expression and the cost of a direct call, saving the cost of an indirect call. When we have a *miss*, though, we pay the cost of the original indirect call and the overhead of evaluating the type-info expression.

The decision whether to transform a call can be based on some heuristics. Unlike the UN optimization, STP involves a penalty (in case of a *miss*). A heuristic can be used to check that the hit-rate is high enough to ensure that we do not pay additional cost over the original indirect call cost (due to *miss* penalties) too often.

2.4 Combining Unique Name and Single Type Prediction

The combination of both UN and STP, denoted by UN&STP, refers to a combined transformation phase, where each UN candidate is transformed to a direct call, and each STP candidate which is not a UN candidate is transformed to a conditional expression.

3 Example and Supporting Results

We demonstrate our ideas on a simple C++ program. The program consists of four files - two *.h* files and two *.C* files:

```
//A.h
class A {
public:
 int data;
 virtual void init();
 virtual void inc() { ++data; }
 virtual void dec() { --data; }
};

extern void foo1(A*);
extern void foo2(A*);
```

```
//B.h
#include "A.h"

class B: public A {
public:
 virtual void dec() {data-=2;}
};
```

```
//A.C
#include "A.h"

void A::init() { data = 0; }
```

```
5
6 void foo1(A* ptrA) {
7  ptrA->A::inc();
8  for(int i=0;i<50000000;i++)
9    ptrA->inc();
10 }
11
12 void foo2(A* ptrA) {
13   ptrA->dec();
14 }
```

```
1  //B.C
2  #include "B.h"
3
4  void main() {
5   A* ptrA = new A,*ptrB = new B;
6   ptrA->init();
7   ptrB->init();
8   foo1(ptrA);
9   foo2(ptrB);
10  for(int j=0;j<50000000;j++)
11    ptrB->dec();
12 }
```

The example program introduces five (indirect) virtual function call sites, two in lines 9 and 13 of *A.C* and the other three in lines 6, 7 and 11 of *B.C*. These are the only calls that use the VFT mechanism. In contrast, the call site in line 7 of *A.C* is a direct virtual function call (fully qualified call).

By examining the assembly code as generated by the C Set ++ compiler for AIX/6000, we can see the difference between the direct and indirect virtual calls. We present here unoptimized code, i.e. code compiled without the -O option.

Direct call in line 7 of *A.C*:

```
7| L4Z      gr3=ptrA(gr1,88)
7| CALL     inc__1AFv,1,gr3...
```

The direct call is translated to two instructions:

1. Load the *this* pointer value.
2. Call function.

Indirect call in line 9 of *A.C*:

```
9| L4Z      gr3=ptrA(gr1,88)
9| L4Z      gr5=(*A).__vfp(gr3,0)
9| L4A      gr4=(*struct {...}).
                    __tdisp(gr5,20)
9| L4Z      gr11=(*struct {...}).
                    __faddr(gr5,16)
9| A        gr3=gr3,gr4
9| CALL     gr11,1,gr3,
              __func__ptr__0...
```

The indirect call is translated here to six instructions:

1. Load the *this* pointer value.
2. Load the address of the appropriate VFT, using the *_vfp* field.
3. Load the *this* pointer adjustment value from the appropriate entry in the VFT.
4. Load the virtual function address from the VFT.
5. Adjust the *this* pointer.
6. Call through pointer to function.

3.1 Unique Name

The UN optimization finds and transforms VFC sites, each of which has a unique implementation. In our example, *A::inc()* is the only implementation of *inc()*, and therefore our tool recognizes the call in line 9 of *A.C* as a UN candidate which is always resolved to *A::inc()*. In order to transform this call to a direct call, we feed our compiler with information on that candidate as follows

A.C, line 9, column 14: A

This indicates the candidate's location and the class in which the single implementation is defined. The compiler front-end then translates the candidate call to a direct call as it does for the call in line 7.

Note that for this UN VFC candidate, the single implementation is defined in *A*, and the call is invoked using a pointer to *A* (*ptrA*). Thus, there is no need to adjust the *this* pointer that points to the beginning of an *A* object. In general, when an adjustment is needed (e.g. when the pointer points to an object of a virtual base class, and the predicted type is a subclass) we save less instructions per transformed location.

Our UN optimization optimizes three calls in the example, and leaves the calls to *dec()* untouched.

3.2 Single Type Prediction

The STP optimization finds VFC sites that are accessible during program execution, and transforms these calls using profiling information. Our STP analysis phase recognizes all five VFC sites in the example as STP candidates, and for each of them the profiling data indicates that all calls are resolved to the same function. For example, all calls in line 11 of *B.C* use *B::dec()*.

For a VFC site, the actual implementation being called depends on the class of the pointed object. Evaluating the *type-info* expression in the optimized VFC should effectively fetch the type of an object at run-time. Since our compiler does not support RTTI [1] (at least at the moment), we use the address of the VFT to which the object points as an indication for its type.

In our example, we feed the STP optimizing front-end compiler with information on the STP VFC candidate as follows

B.C, line 11, column 14: _vft1B1A

This indicates the candidate's location and the VFT which is used most of the times in this VFC. The compiler front-end then generates optimized code, and instead of the original six instructions it translates the candidate call to:

```
11| L4Z       gr3=ptrB(gr1,64)
11| L4Z       gr3=(*B).__vfp(gr3,0)
11| LR        gr4=gr31
11| CL4       cr1=gr3,gr4
11| BT        CL.13,cr1,0x4/eq
11| L4Z       gr3=ptrB(gr1,64)
11| L4Z       gr5=(*B).__vfp(gr3,0)
11| L4A       gr4=(*struct {...}).
                    __tdisp(gr5,28)
11| L4Z       gr11=(*struct {...}).
                    __faddr(gr5,24)
11| A         gr3=gr3,gr4
11| CALL      gr11,1,gr3,
               __func__ptr__2...
11| CL.13:
11| L4Z       gr3=ptrB(gr1,64)
11| CALL      dec__1BFv,1,gr3...
```

The first four lines evaluate the *type-info* expression by comparing the value of $this \rightarrow \_vfp$, the VFT which is pointed by our object, to the address of *_vft1B1A*, the VFT that is used to invoke the predicted target function. Execution will then either branch to the two instructions that stand for the direct call, or else use the six instructions of the indirect mechanism.

[1] run-time type information

3.3 Run-Time Performance Comparison

Finally we illustrate our techniques by comparing execution times of our example program, when compiled with each of our proposed optimizations. In addition, we will illustrate the results when function inlining is enabled, versus those when function inlining is disabled. This will clarify the gain achieved by enabling inline substitutions to virtual function calls, and applying further optimizations. Table 1 shows the execution times for the example program on IBM RISC System/6000, Model 390 (POWER2, 66MHz). Times are measured in seconds. The leftmost column shows the compiler options, where -Q / -Q! mean inlining enabled / disabled, and -O means back-end compiler optimizations enabled. The UN and STP columns presents run times when UN or STP (respectively) optimization is applied on the program. The UN&STP column is the result when first applying UN, and then applying STP where UN cannot be applied.

Options	Original	UN	STP	UN&STP
-O -Q!	25.7	20.3	16.8	15.9
-O -Q	25.6	15.1	13.7	9.1

Table 1: Example run-time measurements

As can be easily seen, when applying both optimizations and inlining, execution time drops from *25.6* to *9.1*. Our example is an extreme case in which the transformation gives room for other global optimizations. The small methods, when inlined and globally optimized, turn into a mere set of few instructions, resulting in a very effective optimized version of code.

4 Implementation

This section describes our implementation work. As stated in Section 2, we develop four independent components:

1. UN analysis component
2. UN transformation component
3. STP analysis component
4. STP transformation component

We also implement a UN&STP transformation component.

The UN (or STP) analysis component generates a list of candidates. This list is the input for the UN (or STP, respectively) transformation component.

The two analysis components are very different. While the UN analysis component requires a static view of the whole program, the STP analysis component requires a dynamic view. The first is achieved by compiling the whole program once, while the second involves executions of an instrumented program in order to collect profiling information.

In contrast, the UN transformation component and the STP transformation component are similar to each other. In both we replace VFC sites by other kinds of expressions offering performance improvement.

Except for the UN analysis component, all other components are in fact variations of the compiler that either instrument the original code, or optimize it. We decided to generate variations of the compiler front-end. Modifying the front-end turns out to be a very comfortable and efficient way to achieve our goals. The UN analysis phase can be accomplished using tools other than the compiler. Even though, our implementation involves yet another front-end variation.

4.1 The Unique Name Analysis Component

We implement a basic algorithm that finds UN VFC candidates. Our implementation does not use any data flow analysis nor class-hierarchy pruning. Thus, we find a subset of all UN candidate call sites. Using a more sophisticated data-collection phase for this optimization may lead to results that are better than the results presented in this paper.

Generating this list is done using a matching-graph. The matching-graph is a bi-partite graph with two kinds of nodes - VFC site nodes, and method implementation nodes. Edges in the graph connect each VFC site to all possible methods that can be called at that site. Building the matching graph consists of:

1. Find all of the VFC sites. Each VFC site is a triplet (*loc, f, A*), where *loc* defines the exact location of this call expression in the C++ source code, *f* is the virtual function name that appears in this call expression, and *A* is a class type name. If the call expression is coded via a class member access, i.e. using dot (.) or arrow (→), preceded by a class object, or a pointer to class object, then the type name *A* is that class. Otherwise, the call is invoked against *this* and it appears in a function member. In that case, the type name is the containing class.
2. Find for a class type *A*, and for one of its virtual functions *f*, all the definitions of *f* that can possibly be called at VFC sites (*loc, f, A*). The list of possible target methods consists of the implementations of *f* introduced by all types derived from *A*, and the implementation of *f* used by *A* (either defined by *A* or inherited from an ancestor of *A*).
3. Connect each VFC site (*loc, f, A*) with all possible methods as found for *A* and *f*. Note that each method definition specifies a class type, namely the type that introduces this definition. Thus, the set of arcs from a particular VFC defines a set of specified types as described in the general type-specification technique.

After building the matching-graph, each VFC site node connected to a single possible target method is included in the list of candidates which is the output of the UN analysis component.

To accomplish Step 2, we use the browser files that are optionally generated by our compiler [14]. A *.pdb* file is created, while *-qbrowse* is specified, for every input C++ file. This is a binary file containing all information that satisfies the C Set ++ browser requirements. By parsing this file, we are able to reconstruct the class hierarchy, and keep track of all virtual methods defined in the program.

The *.pdb* file, however, was found to be unsuitable for Step 1 above. The need to specify a class type with each VFC site, as described above, is too complicated using the *.pdb* structure. In contrast, a very simple variation of the compiler front-end provides this kind of information.

It is worth noting that the global view of the whole program can be obtained using any program data-base or compiler based tool. Our analysis phase could have used other tools such as CodeStore [5] to build the matching-graph.

4.2 The Single Type Prediction Analysis Component

In STP analysis phase we can point out two main stages:

1. Collection of dynamic information (using profiling).
2. Analysis of the above information to result in a candidate list.

In order to collect relevant dynamic information, we implement a special component. This component is a variation of the compiler front-end which transforms each

VFC expression to a comma expression. The comma expression prefixes a call to a special *update* function before the original VFC expression. The *update* function takes two parameters: the address of the VFT used to resolve the call, and the source location of the VFC site. The first parameter cannot be computed at compile-time; it represents the dynamic type of the object against which the call is invoked. Every time control goes through such an instrumented expression, some accumulative variable is updated, recording the fact that a particular VFT is used in the corresponding VFC site.

Running the instrumented program, we collect information on the various VFTs used in each VFC site, and the distribution of calls among these VFTs. When the original program execution ends, the instrumented program writes collective information to binary files.

Summing it up, the phase of dynamic information collection consists of two parts - compiling the program using the special profiling component, and running the program. The result of this phase is the distribution of calls among VFTs in each VFC site. Note that we consider the address of a VFT to be an indication of the type of the object pointing to that VFT.

In the next stage, analysis of the dynamic information, we analyze the output binary file and decide for each VFC site, whether or not to consider it as a candidate site. Also, each candidate VFC site gets associated with some *predicted* VFT. Notice first that since our *type-info* expression relates to a VFT's address, a predicted VFT must be uniquely defined in all compilation units. In other words, a VFT which gets defined in more than one module, so called *duplicated* VFT, cannot serve as a predicted VFT. Other than that, we apply no special heuristic in choosing candidates and predicted VFTs. Every VFC site that is visited at least once during our sample executions is considered to be a candidate VFC site, unless all its associated VFTs are duplicated ones. We choose the most frequently-used unduplicated VFT in that site to be its *predicted* VFT.

4.3 The Transformation Components

Each transformation component is a variation of our compiler front-end, which receives a file containing candidates to be transformed and generates optimized code at the designated places. Each optimizing front-end checks during semantic processing of a VFC site whether additional information regarding this site was supplied as input from the analysis phase. If so, transformation is performed using the name of the *specified* or *predicted* type associated with this candidate.

We implement three transformation components: UN transformation component, STP transformation component and UN&STP one.

The UN transformation component replaces each UN candidate. If the candidate VFC refers to a virtual function *f*, and the specified type is *A*, the optimizing compiler does not generate the original indirect function call expression, but rather a direct call expression to *A::f*.

The STP transformation component replaces each STP candidate. For a candidate VFC that refers to a virtual function *f*, and to some predicted VFT, the optimizing compiler generates a conditional expression <exp> ? <call-exp1> : <call-exp2>. The expression <exp> is an equality expression that compares between $this \rightarrow \_vfp$, the dynamic address of the VFT which is used to resolve the call, and the address of the predicted VFT. The <call-exp1> expression is a direct call, whereas <call-exp2> is the original indirect call expression. The direct call invokes the definition of *f*, the address of which appears in the predicted VFT.

An STP candidate may also be found as a UN candidate. Applying an STP transformation to such a candidate site involves evaluation of <exp>, in addition to the direct invocation in <call-exp1>. The UN&STP transformation component is a combination of UN and STP that prevents this unnecessary overhead. Each UN candidate is replaced by a direct call, and each STP candidate, which is not a UN candidate, is translated to a conditional expression.

4.4 Limitations

Our current transformation components suffer a few limitations:

1. Casting from a virtual base
2. Out of scope VFC sites

Suppose one of our optimizing transformation components processes a VFC site, whose associated class type is *A*, and suppose this VFC is a candidate, the specified (or predicted) type of which is other than *A*, say *B*.

Casting from a virtual base

The direct call involves adjustment of the *this* pointer, as if we were casting *A* to *B*. Currently, no optimization is made if *A* is a virtual base class and *B* is a derived class. Such casting cannot be done at compile-time, thus extra commands have to be generated. We plan to facilitate such casting using data encompassed in VFTs.

Out of scope VFC sites

No optimization is made if *B* cannot be used in the VFC's location, i.e. *B* is not yet defined. Refer to the example in Section 3. The VFC in line 13 of *A.C* corresponds to the base class *A*. This is an STP candidate, the predicted type of which is *B*. But, the class *B* is not defined in *A.C*. This call is not optimized.

We strongly believe that this limitation can be solved within a compiler front-end, by forcing it to refer to *B*'s function members as *extern* symbols. This strengthens our direction to optimize VFCs using compiler variations. An alternative direction, that of single-pass source-to-source translation will remain limited; there is no way in C++ source code to overcome the out of scope problem, unless some code re-arrangement is done in advance (as it is done in [2]).

5 Evaluation

We experiment our implementations against C++ programs that are being used by other groups in IBM as C++ benchmarks [11]. We are examining only those programs that exhibit polymorphism through virtual functions, and can be compiled using the C Set ++ compiler. A brief description of the benchmarks is brought in the following sub-section.

5.1 Benchmarks Used

MetaWare

A performance test created by MetaWare. No input required. The program prints run-time measurements of three different tests. The first two tests construct complicated objects and make heavy use of virtual function calls. The focus of the third test is in applying copy constructors. Obviously, our optimizations significantly affect the first two tests.

Tal-Dict (SOMB)

A performance test of Taligent dictionary class. No input required. The program creates a dictionary of small objects, inserts some objects into the created dictionary, and makes a lot of look-ups into the dictionary. Since none of the methods are inline functions, we consider this program a good benchmark to test the improvement gained without inlining.

LCOM

A compiler for the hardware description language L. We used one of the inputs supplied with the distribution package (circuit2.l).

5.2 Profiling Results

To evaluate the potential cost and benefit of our optimizations, we use a set of measurements:

1. Function call sites

   We count all call sites in the user-compiled code including compiler-generated calls (and calls within compiler-generated functions), such as constructors and destructors. Files are compiled with no inlining and no optimization. For example, *B.C* in our running example includes ten function call sites: four are used to define *ptrA* and *ptrB*, five are explicitly coded in lines 6, 7, 8, 9 and 11. Another call appears in the default compiler-generated constructor for *B*. The latter calls *A*'s constructor.

2. Active function calls

   Number of function calls performed during a program execution. We count only those calls invoked from sites referred in item 1 above. In our example we have 100,000,011 active function calls.

For the above two items we implemented a profiling tool, based on a compiler front-end variation, that instruments the code. Running this instrumented program yields the required numbers.

3. VFC sites

   Number of all call sites in the user code where unqualified virtual function call expression is coded. In our example there are 5 VFC sites.

The next seven measurements are computed using the STP analysis component. All numbers refer to a specific program execution. Recall that the STP analysis component computes distribution of calls among unduplicated VFTs in each VFC site. Thus, for each VFC site, *i*, we get n_i numbers, each of which represents the number of times a particular VFT is used.

4. Active virtual function calls

   Number of calls corresponding to VFC sites performed during a program execution, and its percentage out of all active function calls. For our example we get 100000003 active virtual function calls.

5. Unaccessible VFC sites

   VFC sites that are never reached in a program execution. In our example, there are no unaccessible VFC sites.

6. Virtual calls using a single table

   Summing up all numbers that correspond to VFC sites for which $n_i = 1$, and the percentage out of all active virtual function calls. In our example, each VFC site uses a single table.

7. Virtual calls using two tables

   As before with $n_i = 2$.

8. Virtual calls using more than two tables

   As before with $n_i > 2$.

9. Virtual calls using most frequent table

   The number we get from summing the number of calls taken through the predicted table (i.e. the most frequently used VFT) in each candidate VFC site, and the percentage out of all active virtual function calls. This is the total number of calls that will turn to be direct calls after our STP transformation.

10. Virtual calls using table with >50% frequency

    As in the previous item, but referring only to those VFC sites for which the predicted table is used more than in 50% of the cases.

The last two items are obtained using the UN analysis component, and profiling information.

11. Virtual calls with unique inline implementation

    Number of calls corresponding to UN VFC candidate sites performed during a program execution, and its percentage out of all active virtual function calls. We consider here only those candidates for which the single implementation is an inline function.

12. Virtual calls with unique non-inline candidates

    As in the previous item, but the single implementation is not an inline function.

Table 2 presents the results of these profiling measurements on our benchmarks.

5.3 Run-Time Performance

Run-time measurements for our benchmarks can be found in Table 3. Files are compiled with inlining (-Q) and back-end optimizations (-O). Times are measured in seconds on IBM RISC System/6000, Model 390.

Results for MetaWare show an improvement of 20%. Although the percentage of virtual functions in the benchmark is relatively low, the inlining that follows the transformation makes the improvement extremely significant. Note that all VFC sites in MetaWare are UN candidates, and hence the run-time result after applying UN&STP is identical to that achieved by UN optimization alone. When applying STP optimization to MetaWare, the cost of evaluating the conditional expression causes the improvement to be less significant than the one gained by UN. The first two tests in MetaWare make heavy use of the virtual function call dispatching mechanism. Run times for MetaWare's first two tests are brought on a separate line to emphasize that opportunities for transformation are local to some part of the program, in which improvement of 43% is reached.

For Tal-Dict, improvement performance achieved by STP is better than the one achieved with UN since only 41% of the VFC sites are UN candidates. When combining UN&STP , we reach a run time of 11.4 seconds, which is a 20% improvement of the original run time.

Poor results for LCOM are due to the low percentage of active virtual function calls in the program. In addition to that, only about 1% of them correspond to UN candidates.

It is worth noting that all of the results brought here were achieved using our limited transformation (see Section 4.4). In fact, we are unable to transform about 25% (average) of all VFC sites. Better results are expected in future implementations of our components.

6 Summary

The C++ community recognizes the importance of C++ performance, and turns to be more and more conscious about the need to enhance run-time results. Given that many of C++ users come from the arena of C programming, and given the well-established notion of compiler optimization in C, it is very much expected that C++ compilers are to put more effort on code optimization. The fact that C++ programs perform indirect calls more often than C programs is one of the most significant dynamic performance characteristics of C++ versus C. Most indirect branches are caused by virtual function calls. This all

	MetaWare	Tal-Dict	LCOM
Function call sites	60	1706	2871
Active function calls	74000030	53125698	5831087
VFC sites	5	80	455
Active virtual function calls	5000000 6%	35060980 66%	1100949 19%
Unaccessible VFC sites	0	67	154
Virtual calls using single table	5000000 100%	35060980 100%	569218 52%
Virtual calls using two tables	0 0%	0 0%	14903 1%
Virtual calls using more than two tables	0 0%	0 0%	516828 47%
Virtual calls using most frequent table	5000000 100%	35060980 100%	1071925 97%
Virtual calls using table with >50% frequency	5000000 100%	35060980 100%	1055810 96%
Virtual calls with unique inline implementation	5000000 100%	0 0%	5443 0.5%
Virtual calls with unique non-inline implementation	0 0%	14321318 41%	4822 0.4%

Table 2: Profiling results for benchmarks

	Original	UN	STP	UN&STP	Performance improvement
MetaWare	6.2	5.0	5.6	5.0	20%
MetaWare's first two tests	2.8	1.6	2.1	1.6	43%
Tal-Dict	14.3	12.4	12.3	11.4	20%
LCOM	3.7	3.65	3.55	3.55	4%

Table 3: Run-time measurements for benchmarks

brings us to our strong belief that optimizing C++ virtual function calls is one of the most challenging directions in C++ compiler optimizations.

As shown in this article, replacing indirect virtual function calls by direct calls or conditional expressions seems to be a very promising solution to reduce the cost of these calls. Our main contribution is in introducing techniques that support the whole spectrum of code analysis and code transformation. The fact that we develop variations of a production compiler makes the opportunity of introducing such improvements into a real compiler firm and realistic.

Still, there is much room for further investigation. In order to better assess our contribution we intend to apply our techniques on additional benchmarks. We are now communicating with other groups to accept additional C++ programs that may serve as benchmarks for our optimization work. We are facing the following problems:

1. Language compatability. Many programs do not get compiled with our version of the C Set ++ compiler.
2. Object-orientation style. Many programs do not explore polymorphism to the extent that make them good candidates for our techniques.
3. Size. Some interesting real programs are too heavy and large to be examined and processed under our limited development resources. An example of such a system is described in [6].

More experiments will facilitate a thorough analysis of the cost and benefit of our transformations, as well as evaluating the significance in introducing inlining opportunities. We plan to apply techniques to measure the stability of our type-profiling information, similar to what is done in [9].

There are many papers that focus on the UN analysis phase. We plan to communicate with some leading researchers in this area, and make serious comparisons between their work and ours. Of particular interest is a recent work by Bacon and Sweeney ([4]).

More implementation effort should be put to eliminate current limitations of our transformation tools (see Section 4.4). We also plan to implement the general *type-specification* and *type-prediction* optimizations. Obviously enough, the main effort in carrying out these generic optimizations is expected to be exactly at that point that was overly simplified in the special cases, UN and STP, namely the need to apply heuristics to decide on the candidate VFC list. The process of evaluating cost and benefit becomes very complicated in this case.

References

[1] O. Agesen and U. Holzle. Type Feedback vs. Concrete Type Inference: A Comparison of Optimization Techniques for Object-Oriented Languages. In *OOPSLA '95*, Austin, Texas, 1995.

[2] G. Aigner and U. Holzle. Eliminating Virtual Function Calls in C++ Programs. Technical Report TRCS 95-22, Dept. of Computer Science, University of California, Santa Barbara, December 1995.

[3] D.F. Bacon, M. Burke, G. Ramalingam, H. Srinivasan, and P.F. Sweeney. Compiler Optimizations for C++. Technical report, IBM Thomas J. Watson Research Center, March 1995.

[4] D.F. Bacon and P.F. Sweeney. Rapid Type Inference in a C++ Compiler. Technical report, IBM Thomas J. Watson Research Center. Submitted to PLDI'96, 1995.

[5] J. Barton, P. Charles, Y.-M. Chee, M. Karasick, D. Lieber, and L. Nackman. Codestore: Infrastructure for C++ - Knowledgeable Tools. In *OOPSLA '94*, 1994.

[6] W. Berg, M. Cline, and M. Girou. Lessons Learned from the OS/400 OO Project. *Communications of the ACM*, 38(10):54–64, October 1995.

[7] B. Calder and D. Grunwald. Reducing Indirect Function Call Overhead in C++ Programs. In *Conference Record of the Twenty-First ACM symposium on Principles of Programming Languages (POPL)*, pages 397–408, Portland, Oregon, January 1994. ACM Press, New York, New York.

[8] B. Calder, D. Grunwald, and B. Zorn. Quantifying Behavioral Differences Between C and C++ Programs. Technical Report CU-CS-698-94, University of Colorado at Boulder, January 1994.

[9] D. Grove, J. Dean, C. Garrett, and C. Chambers. Profile-Guided Receiver Class Prediction. In *OOPSLA '95*, Austin, Texas, 1995.

[10] U. Holzle and D. Ungar. Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback. In *SIGPLAN '94 Conference on Programming Languages Design and Implementation*, pages 326–336. ACM Press, June 1994.

[11] Y. Lee and M.J. Serrano. Dynamic Measurements of C++ Program Characteristics. Technical Report TR ADTI-1995-001, IBM, January 1995.

[12] H.D. Pande and B.G. Ryder. Static Type Determination for C++. In *Proceedings of the Sixth Usenix C++ Technical Conference*, pages 85–97, April 1994.

[13] J. Plevyak and A. Chien. Type Directed Cloning for Object-Oriented Programs. Technical report, University of Illinois at Urbana-Champaign, www-csag.cs.uiuc.edu/individual/jplevyak, 1996.

[14] IBM Publication. *IBM C Set++ for AIX/6000: User's Guide*, 1993.

Composing Special Memory Allocators in C++

Keith Loepere[1]

Open Group Research Institute
loepere@osf.org

The nature of complex, high performance system software imposes constraints on the nature and operation of memory allocation or requires special properties to be true for the memory. Often multiple constraints or properties must hold at once. It is desirable to express these constraints and properties as classes and methods. This paper presents a technique for expressing and composing special memory allocation mechanisms in C++.

Introduction

The Open Group Research Institute recently completed the initial version of MK++ [5], an operating system microkernel implemented in C++, based on the concepts that underlie the Mach operating system. The long term objective of the MK++ project is to produce a high performance, scalable microkernel technology that supports real-time computing and provides a high level of assurance; indeed, the current implementation of MK++ is being used as the basis for a highly secure operating system environment, while providing excellent performance.

The nature of a microkernel's software imposes considerable constraints on the use of memory — how it is laid out, how usage is accounted, the conditions under which allocation is performed, and so on. In the past, OS's provided and satisfied these constraints through code scattered throughout the system, at the points where such storage is allocated, used and de-allocated. Since MK++ was being written in C++, with extensive object orientation, it seemed natural to encapsulate the implementation of these special memory mechanisms within classes and methods, especially the C++ language defined memory allocation operators.

Most examples in the literature of the use of the C++ ability to define special memory allocation ([6] being the best tutorial) concerns using type specific knowledge to optimize allocation, e.g. pool allocation. There are many other reasons why one would want to have special memory allocators:

- support for variable sized objects
- data caching
- storage recycling
- allocation from special (physical) memory areas
- dynamic grouping
- real-time memory management
- memory resource control
- debugging

This paper starts with a discussion of some of these special allocation needs as motivation. The body of the paper presents a technique by which these allocation needs can be expressed and a technique for composing special memory allocation mechanisms in C++.

Special Allocation Needs

Complex system and application software has many needs for special memory allocators. Some are considered here.

Buffer Objects

The first example concerns an object that has an associated variable sized buffer. The C trick:

```
struct foo {...; char buf[1];};
```

(in which additional space is allocated beyond the declared end of the structure as an "extension" of `buf`, and in which the user will explicitly override the subscript of `buf`) does not work in C++. The compiler cannot be relied upon not to place fields past the declared end of a class, especially in the face of derivation. Without the use of special allocation, an object with a variable sized buffer would have to be allocated in two pieces — the object and the buffer —

1. This research was supported in part by the Advanced Research Projects Agency (ARPA) and the Air Force Materiel Command (AFMC). Copyright 1996 by the Open Group Research Institute. All Rights Reserved

requiring two calls to the underlying storage allocator. With the use of a special allocator, these pieces can be allocated as one unit.

Debugging

An obvious place to place memory allocation debugging logic is in special allocators — e.g., allocating extra storage as guard areas, keeping redundant size information and providing memory tracing.

Pool Allocation

Pool allocation refers to a range of specialized allocation strategies in which separate logical memory areas are established for allocation of some subset of the objects in the system. A common property of these strategies is that some class-specific knowledge is used to optimize allocation. The most common pool allocation strategy divides a memory pool into fixed sized chunks, where the chunk size is the size of some class of objects. Allocation of one those objects becomes a simple matter of finding a free chunk.

Pool allocation can also be used for allocation of objects of disjoint classes for which some common storage property must hold. In real-time processing, dedicated areas may be set aside for allocations that must not fail, or that must complete in bounded time [7]. Another example is the allocation of objects in shared memory [3] where the particular region needs to be specified. Unlike the earlier examples above, in which the identity of the pool can be implicit in the class of objects involved, in these latter examples the identity of the pool must be explicitly passed to the special allocator (by overloading the new operator).

A related idea is recyclable storage, in which there are *recycling center* objects that hold onto some number of storage units that were previously allocated so as to reduce the cost of subsequent allocations. The idea is similar to pool allocation, except that no special area need be set aside up front for the objects' storage, and only a limited amount of storage is allowed to be idle.

Per-Thread Caching

In a multi-threaded environment, a shared memory pool or recycling center requires some form of concurrency control (such as locking) to provide safety of the allocation and de-allocation. When high performance is needed, one can avoid this synchronization overhead by caching some storage against each thread, via a special allocator.

Dynamic Grouping

Traditional virtual memory systems often perform poorly when supporting large object systems because of the typically small object size and potentially far flung allocation of related objects. For a consideration of the issues, [8] discusses the idea of dynamic grouping of objects, [4] discusses the development of application-specific paging mechanisms, and [1] can be thought of as a combination.

Special allocators have two roles to play in this area. First of all, they can encapsulate the mechanisms that would make allocation time decisions as to the placement of objects based on some understanding of the usage patterns. Secondly, special allocators can accumulate allocation pattern information, perhaps for making dynamic placement decisions, or possibly for off-line analysis to be fed into developing pre-optimized placement strategies.

Memory Resource Control

A server maintaining state on behalf of multiple clients may face problems managing its memory resources. As part of making fairness decisions, or considering load distribution, or generally assessing the burden of the various clients, it can be desirable to keep some knowledge of the memory resources consumed by various activities. Implementing such accounting in special memory allocators makes the accounting automatic and centralizes the logic involved in maintaining the information.

Language Defined Memory Allocators

This section reviews C++'s support for dynamic memory allocation; an understanding of the language's more subtle aspects with respect to this functionality will simplify the discussion that follows. (Refer to [2] for a complete exposition.)

In C++, one dynamically (heap) allocates an object of class T as follows:

```
    T * Tp = new(Nargs) T(Cargs);
```

This statement causes two functions to be invoked (in this order):

```
    T::operator new(sizeof(T), Nargs)
    T::T(Cargs)
```

That is, an operator new function is called, that defined for (or inherited by) class T[2], given the size of the object

2. Or the global operator if no class operators are defined.

to allocate[3] along with additional arguments. This operator allocates the "raw" storage to hold the object. An appropriate constructor is then called to initialize the storage.

Conversely, the de-allocation of a dynamic (heap) object, given a pointer to it:

```
    delete Xp;
```

causes the following two functions to be called (in this order):

```
    T::~T()
    T::operator delete(void *)
```

That is, the destructor for type T is called (with no arguments) followed by a call to an operator delete, given a pointer to the storage, which is expected to deal with the storage. Assuming the destructor is virtual[4], the operator delete called is that defined for (or inherited by) the most derived class with a destructor.

The rules of C++ significantly restrict the way in which special allocators can be implemented.

The pointer returned by operator new references the raw storage for the object; given the permission the language gives to the compiler to lay out that storage, only the most derived class has the property that the raw storage pointer points to itself. Given that one can derive new classes from a class T that defines an operator new, `T::operator new` and `T::T` do not inherently know the relationship between the raw storage and the base class storage. That is, these two operators cannot pass information between them through storage without extra help.[5]

Likewise, the pointer passed to any given destructor is not necessarily that passed to operator delete.

Although one can define multiple operator new's to define multiple memory allocation behaviors, the impracticality of having the single operator delete coping with all of those behaviors leads one to encapsulating different behavior via different operator new / operator delete pairs. In any case, since one cannot pass additional arguments to the destructor or operator delete, they need extra help to have all the information they need lying around.

The next section introduces a stylized and composable technique for providing this "extra help."

Generalizing Allocation Information

One of the key goals of the structured special memory allocators is to encapsulate the special memory behavior in memory operators. To be able to define a new operator new / operator delete pair, one must declare a new class, so it is really a class that encapsulates the special memory behavior. However, given a class that defines special memory behavior, it is desirable to be able to derive other classes from it, without those classes needing to be aware of the special memory allocators. Since each of the operator new, constructor, destructor and operator delete methods for the special memory class will be provided with different information such that they do not know, or have consistent knowledge, of the layout of the overall object being allocated, the object itself does not serve to hold the special information that operator new and operator delete will use. Thus, this extra information must be kept outside of the object proper.

For performance, the allocation of a special memory object should not involve two calls to the underlying heap allocator; the object storage and the allocation information want to be kept together.

This special allocation information is here called the *allocation record*. As described in more detail below, allocation records derive from the base class `allocrec` and objects allocated with allocation records derive from the base class `allocation`. The job of `allocation::operator new` is to arrange for the allocation of sufficient storage for the object proper and its allocation record, get the allocation record initialized, and generally arrange the storage so that subsequent constructors, methods and finally operator delete can find the components. Figure 1 shows the basic allocation layout as arranged by `allocrec` and `allocation`. The reason for this layout is coupled to the mechanics of how it is established and used.

Since the special allocation properties are associated with the allocation record information, that record is a class in its own right (`allocrec`) to which class `allocation` delegates to provide the mechanics. `allocation::operator new` delegates to the static method `allocrec::alloc`, which allocates space for the allocation record, a "back pointer" and

3. This is the size of the most derived class, even if the most derived class does not define these special operators.
4. Otherwise the operator delete is chosen by the static type of Xp.
5. With `dynamic_cast<void *>`, a base class constructor can locate the beginning of the most derived class (the raw storage), but the converse is not true — operator new, knowing the location of the raw storage, cannot reasonably locate where any particular base class will lie within that, nor can safely store data there.

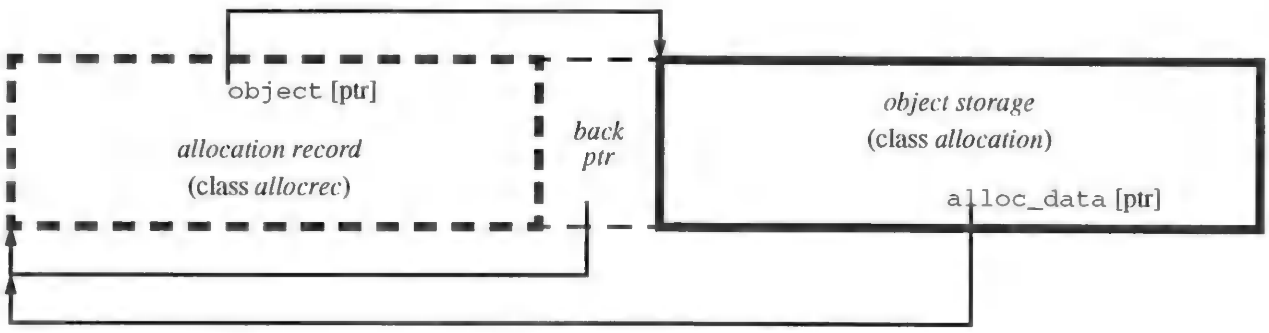


Figure 1 Object Allocation Layout

then space for the object proper. The back pointer enables `allocation::allocation` and `allocation::operator delete` to find the allocation record. `allocation::allocation` can find the back pointer (in the presence of derivation from class `allocation`) by locating the beginning of the most derived class (with `dynamic_cast<void *>`) and passing that pointer to `allocrec::record`. `allocation::allocation` saves the pointer to the allocation record (in `alloc_data`) so that methods may efficiently access the allocation information. Definition 1 defines class `allocation`.

```
class allocation {
public:
   allocation(): alloc_data(allocrec::record(dynamic_cast<void *>(this))) {}
   void * operator new(size_t obj_size) {
      return allocrec::alloc(obj_size);
   }
   void operator delete(void * item_base) {
      allocrec::record(item_base)->discard();
   }
   virtual ~allocation() {}
protected:
   allocrec * const alloc_data;                  // allocation record base
};
```

Definition 1 Class allocation

`allocation::operator delete` delegates to class `allocrec` via the virtual `discard` method to free or otherwise process the storage. When `allocation::operator delete` is called, the allocation object will have been destructed, and the pointer passed to operator delete will locate the raw storage, as opposed to the `allocation` class (in the presence of derivation). As such, operator delete needs to reference through the back pointer (by delegating to class `allocrec`).

Definition 2 defines class `allocrec`. Note the overloaded operator new so as to receive the size of the object itself (since the compiler would only pass the size of the allocation record). Note that the back pointer points to the `allocrec` base class (as opposed to the most derived allocation record class). As such, `allocrec::operator new` (which doesn't know the layout of the most derived allocation record) can't set the back pointer; only `allocrec::allocrec` can. `allocrec::allocrec` saves a pointer to the raw storage for the object proper in field `object`.

The `allocrec` class accepts three size parameters: the total size of the allocation record, the total size of the object proper, and an "extra" size. The purpose of this third size is for extra space allocated between the allocation record and the (back pointer before the) object. The code assumes that the caller ensured that the "extra" size preserves pointer alignment by being a multiple of the size of a pointer. (The code shown assumes that the alignment requirements for pointers are the strictest of that for any fundamental type.)

```
class allocrec {
public:
   allocrec(size_t rec_size, size_t extra_size):
      object(((char *)dynamic_cast<void *>(this))+ rec_size+ extra_size+
         sizeof(allocrec *)) {
      *(((allocrec **)object)-1) = this;        // set back pointer
   }
   static void * alloc(size_t obj_size) {       // "allocate"
      return (new(obj_size, 0) allocrec())->object;
   }
   virtual void discard() {                     // "de-allocate"
      delete this;
   }
   void * operator new(size_t rec_size, size_t obj_size, size_t extra_size) {
      return malloc(rec_size+ obj_size+ extra_size+ sizeof(allocrec *));
   }
   void operator delete(void * alloc_base) {
      free(alloc_base);
   }
   static allocrec * record(void * item_base) {
      return *(allocrec **)(((char *)item_base)- sizeof(allocrec *));
   }
   virtual ~allocrec() {}
   void * const object;                         // most derived object class
protected:
   allocrec(): // virtual bases require this definition
      object(((char *)dynamic_cast<void *>(this))+ sizeof(allocrec)+
         sizeof(allocrec *)) {
      *(((allocrec **)object)-1) = this;
   }
};
```

Definition 2 Class allocrec

Composing Special Allocation Properties

This section demonstrates by example how to encapsulate and compose special allocation properties by deriving from the classes `allocrec` and `allocation`.[6]

Buffer Object

With the use of a special allocator, an object with a variable sized buffer can have its two pieces — the object and the buffer — allocated as one unit. Figure 2 shows the memory layout. The buffer is kept in the "extra" space allotted by the `allocrec` class.

6. The examples are representative of special allocators in MK++. Various details of the actual MK++ microkernel environment are omitted; as such, these examples are a simplification and yet a generalization of the MK++ allocators.

Definition 3 defines an allocation record supporting an arbitrary sized buffer. The `alloc` method shows the typical form — an overloaded new of that type's allocation record with extra space set aside for the object and buffer. The constructor assumes that this class is the only one that requests and uses the "extra" space. A more general buffer allocator could subdivide the extra area as indicated by derived allocators.

Definition 4 shows the object definition. The operator new has the typical form — simple delegation to the allocation record type. For efficiency of later access, the constructor fetches and saves a pointer to the buffer area. Note, as shown in Definition 5, how derivation is possible from class `buffer` with no special concern.

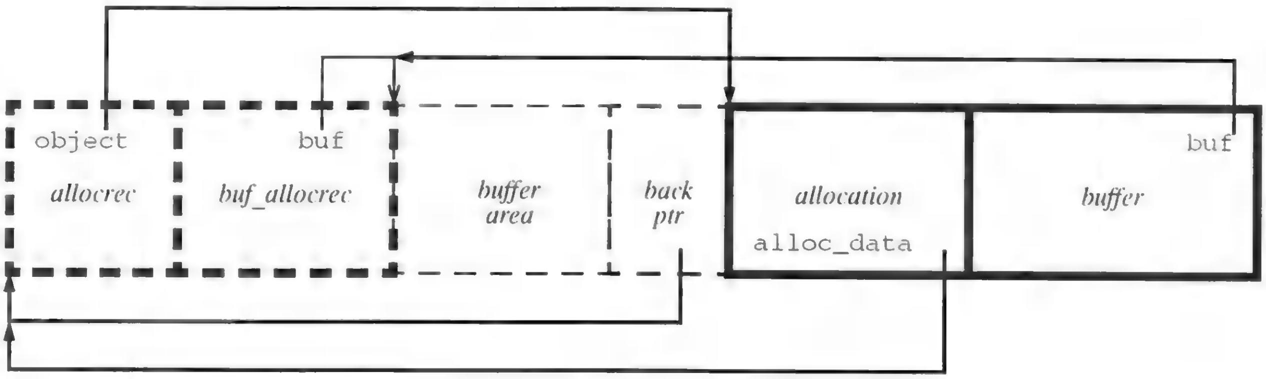


Figure 2 Buffer Object Allocation Layout

```
class buf_allocrec: public virtual allocrec {
public:
   buf_allocrec(size_t rec_size, size_t buf_size): allocrec(rec_size,
         buf_size), buf(((char *)dynamic_cast<void *>(this))+rec_size) {}
   static void * alloc(size_t obj_size, size_t buf_size) {
      return (new(obj_size, buf_size)
         buf_allocrec(sizeof(buf_allocrec), buf_size))->object;
   }
   void * const buf;
};
```

Definition 3 Class buf_allocrec

```
class buffer: public virtual allocation {
public:
   buffer(): allocation(),
      buf(dynamic_cast<buf_allocrec *>(alloc_data)->buf) {}
   void * operator new(size_t obj_size, size_t buf_size) {
      return buf_allocrec::alloc(obj_size, buf_size);
   }
   void * const buf;
};
```

Definition 4 Class buffer

```
class message: public buffer {
public:
   message(destination & a_target):
      allocation(), buffer(), target(a_target) {}
protected:
   destination & target;
};
```

Definition 5 Class message

Recyclable Storage

In this example, a *recycling center* (class `recycler`) is assumed to have a `get` method, which returns an unconstructed (raw) storage unit, if one is available, and a `put` method, which accepts a destructed (raw) storage unit (unless doing so would exceed some recycling policy, in which case the storage will need to be released).

The design decision here was to make recycling an allocation property rather than an object property. If recycling were an object property, than it would be done against a constructed object, most likely leaving it constructed. By making it an allocation property, recycling is done by operator new and operator delete, against raw storage either never constructed or subsequently destructed.

The reason for this decision (aside from purity) is that it allows recycling to occur without the object's user's knowledge; the user can simply "delete" the object when done, and operator magic recycles the storage for a subsequent "new". Note, though, that when the user "deletes" the object, it is destructed, but the allocation record is not. The allocation record is destructed only when the storage is to be de-allocated.

Figure 3 shows the allocation layout. Definition 6 defines the allocation record, which contains a reference to the target recycling center. The `alloc` method differs from the typical form in that it first asks the recycling center for storage before allocating some. Likewise, the `discard` method is redefined to try recycling before de-allocating the storage. Definition 7 defines the `recyclable` class.

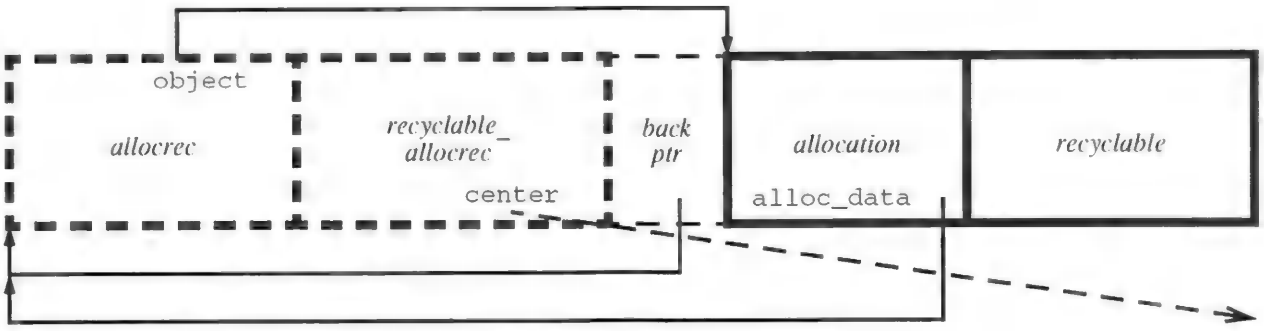


Figure 3 Recyclable Object Allocation Layout

```
class recyclable_allocrec: public virtual allocrec {
public:
   recyclable_allocrec(size_t rec_size, size_t extra_size, recycler &
         owner): allocrec(rec_size, extra_size), center(owner) {}
   void discard() {
      if (! center.put(object))
         delete this;
   }
   static void * alloc(size_t obj_size, recycler & owner) {
      void * target = owner.get();              // see if something recycled
      return target? target: (new(obj_size, 0)
         recyclable_allocrec(sizeof(recyclable_allocrec), 0, owner))->object;
   }
   recycler & center;
};
```

Definition 6 Class recyclable_allocrec

```
class recyclable: public virtual allocation {
public:
   recyclable(): allocation() {}
   void * operator new(size_t obj_size, recycler & owner) {
      return recyclable_allocrec::alloc(obj_size, owner);
   }
};
```

Definition 7 Class recyclable

Combining Allocators

The above allocators can be combined to provide a recyclable message object through multiple (virtual) inheritance.

Figure 4 shows the storage layout. This example emphasizes the reason why the pointers are established the way they are. Note that the `alloc_data` pointer does not point to the start of the allocation record, and so it is not possible to locate the buffer area if its location had not been recorded. Likewise, class `allocation` is not at the start of the most derived object, so the extra space cannot be located via its this pointer.

Definition 8 defines the allocation record class. The methods are a simple composition of the base classes. Note that `alloc` must be redefined, so as to allocate the appropriate allocation record, but `discard` need not be redefined — that from `recyclable_allocrec` is fine.

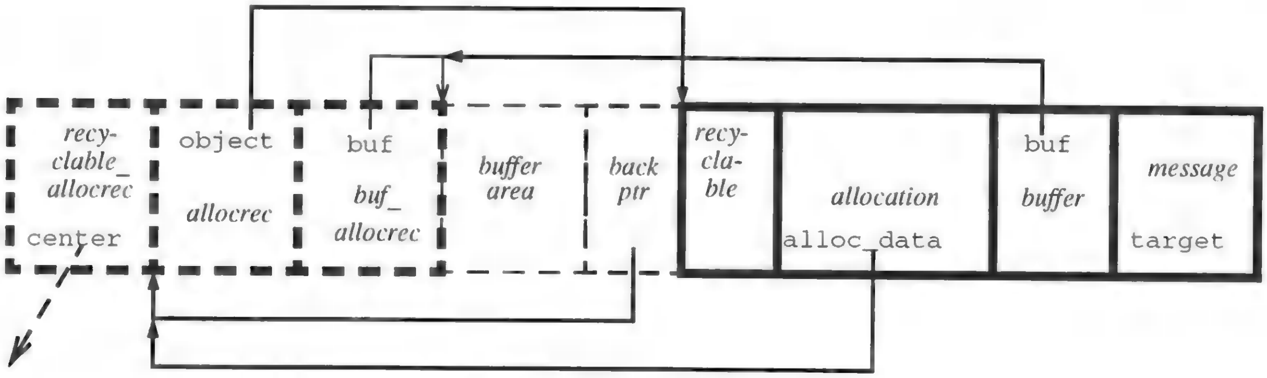


Figure 4 Recyclable message storage layout

```
class recyclablebuf_allocrec: public recyclable_allocrec, public buf_allocrec {
public:
   recyclablebuf_allocrec(size_t rec_size, size_t buf_size, recycler & owner):
         allocrec(rec_size, buf_size), buf_allocrec(rec_size, buf_size),
         recyclable_allocrec(rec_size, buf_size, owner) {}
   static void * alloc(size_t obj_size, size_t buf_size, recycler & owner) {
      void * target = owner.get();              // see if something recycled
      return target? target: (new(obj_size, buf_size)
         recyclablebuf_allocrec(sizeof(recyclablebuf_allocrec), buf_size,
         owner))->object;
   }
};
```

Definition 8 Class recyclablebuf_allocrec

```
class recyclable_message: public recyclable, public message {
public:
   recyclable_message(destination & a_target): allocation(),
         message(a_target), recyclable() {}
   void * operator new(size_t obj_size, size_t buf_size, recycler & owner) {
      return recyclablebuf_allocrec::alloc(obj_size, buf_size, owner);
   }
};
```

Definition 9 Class recyclable_message

Definition 9 defines the `recyclable_message` class, which has the typical form. Note that each derivation of the allocation record had a corresponding derivation of the object class, but the converse is not true. Specially allocated objects can inherit without affecting the storage allocators.

MK++ Evaluation

The technique presented in this paper is used extensively within the MK++ microkernel.

Uses

--- Buffer Objects

Aside from their use for messages, MK++ uses buffer objects for physical memory descriptors. The object proper provides the threading of the descriptor into an address space or message, and the buffer area holds an arbitrary sized array of physical page locators.

--- Storage Recycling

MK++ uses recyclable storage to support network input packet processing. When a network packet arrives, a packet object is selected to hold the data. Since the general system memory allocator can block waiting for availability of memory, and network input processing takes place in interrupt context where blocking is not permitted, this allocation must take place from a non-blocking, pre-allocated pool. The purpose of using recyclable storage has to do with the fact that the thread that subsequently processes the packet object does not know it is a packet object, only that it is some form of message. By using recyclable storage, the "deletion" of the message object arranges for the storage to be returned to the network input packet pool, not only replenishing the pool, but also self-throttling the maximum delivery rate of packets to the rate of packet processing. A memory pool with special resource management policies is also a candidate for storage recycling.

--- Memory Resource Control

An optional feature of MK++ is space accounting — resource management mechanisms that keep an accounting of memory utilization and take action when memory usage by a resource account exceeds reasonable limits. This feature is implemented in one of the most base classes in the allocator derivation hierarchy, providing for automatic accounting of storage. By design, each data structure in the kernel is considered to be "owned" by a specified high level system object (a "space source"). All allocations require a space source to be named (to operator new) which tracks the total memory consumption.

Space accounting is normally only enabled for environments in which uncontrolled memory usage by untrusted tasks may occur. However, it has found considerable use as a memory leak detector, asserting system failure when a space source is deleted without having de-allocated all its associated sub-structures.

Performance

The code shown in this paper was purposely coded using virtual base classes and virtual methods to show generality, and to show that the techniques work in the presence of those mechanisms. The uses within MK++, however, do not use virtual base classes or methods[7] and the code is extensively inlined.

An object allocated with an allocation record consumes additional space. It is assumed, for objects for which these techniques are needed, that the extra data in the allocation record (buffer information, for example) would be in memory somewhere anyway, so the real overhead is the various pointers linking the information. In the absence of this technique, the allocation information and the object proper would be allocated as two separate pieces with some sort of linkage, and the extra allocation unit is likely to have allocation overhead from the underlying system allocator, so the memory consumption balances out.

The ultimate test of the suitability of this technique for MK++ was the ability of the compiler (gcc in our case) to optimize it. For MK++' most important case, `new cached_message()` (determine the current thread, locate its cached storage and construct a message object in it) takes 11 instructions on an Intel 486, and the cost of the corresponding delete (to destruct the object, determine the current thread and return the cached storage to it) takes 10 instructions, each of which is barely more than the cost of the calling sequence to the system's allocator, yet alone the cost of executing any code within it.

Summary

The use of special allocators provides the structure by which complex memory allocation properties and mechanisms can be encapsulated and composed. The technique described in this paper extends the C++ inheritance mechanisms to the area of memory allocators. These composed allocators allow complex allocation properties to be encapsulated, not only hiding their details and removing their concerns from the users of the objects, but also allowing high performance memory mechanisms to be easily implemented.

References

[1] Jeffrey S. Chase, Henry M. Levy, Edward D. Lazowska and Miche Baker-Harvey,

7. Virtual base classes and methods are used within MK++, but not currently within allocation record classes.

Lightweight Shared Objects in a 64-Bit Operating System, Proceedings OOPSLA '92, ACM SIGPLAN Notices, 1992.

[2] Margaret A. Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison Wesley, 1990.

[3] David Jordan, *Instantiation of C++ Objects in Shared Memory*, Journal of Object-Oriented Programming, Vol 4, No 1, 1991.

[4] Keith Krueger, David Loftesness, Amin Vahdat and Thomas Anderson, *Tools for the Development of Application-Specific Virtual Memory Management*, Proceedings OOPSLA '93, ACM SIGPLAN Notices, 1993.

[5] MK++ Project, MK++ Kernel High Level Design, Open Group Research Institute, 1996.

[6] Robert B. Murray, C++ Strategies and Tactics, Addison Wesley, 1993.

[7] Franco Travostino, Franklin Reynolds and Tim Martin, *Real-Time Local and Remote MACH IPC: Architecture and Design*, Operating Systems Collected Papers, Vol. 3, Open Software Foundation, 1994.

[8] Ifor Williams, Mario Wolczko and Trevor Hopkins, *Dynamic Grouping in an Object-Oriented Virtual Memory Hierarchy*, Proceedings ECOOP '87, Springer-Verlag, 1987.

Building Independent Black Box Components in C++

Mark Addesso
Software AG of North America

Abstract

When using Object Oriented techniques to build systems, the relationships between objects create dependencies which inhibit reuse, unit testing and reliability. By using black box components, these relationships can be factored out of the object to create truly independent components. Techniques to design and build systems in this manner are presented. A complex system built with black box components is also described.

Introduction

The basic idea with object technology was to create packaged units of data and behavior. You could put your hands around an object, design and code it separately, test it as a unit. Once completed, this package of functionality could be reused and would never have to be coded again.

It sounds good in theory, but in practice it is very difficult to create such independent packages. The difficulty is that any complex behavior requires a group of collaborating objects. The boundaries of these packages get blurred as one object requires the services of another, or needs to query another object, or starts to control another object. When the boundaries get blurred, all the advantages of the packaged behavior and data blur as well. Objects which depend on others cannot be built and tested independently, and further, cannot be reused without including all the dependent objects.

These dependencies are inherent in current object design techniques. Most popular techniques let you model interactions between objects. Wirfs-Brock uses Collaboration Graphs [1], Rumbaugh uses Event Trace Diagrams [2], Jacobson uses Interaction Diagrams [3], etc. These interaction may be described as stimuli to another object, a contract between two objects, etc. Any direct interaction between objects creates a dependency. The object sending the message must contain a reference to the receiving object, and must know the exact protocol of the receiving message.

To reuse an object with dependencies in another application, the receiver must also be included, and the application must correctly establish the relationship between the two. Even with two objects, reuse becomes complicated. Given all message sends of an object, it may be dependent on many objects. Even worse, the dependent objects, can be dependent on others, etc., etc. Thus, the entire web of connected objects would have to be included and correctly initialized to be reused in another application. This is why currently only primitive classes (lists, widget, etc.) or large subsystems (OLE components) are good candidates for reuse.

Removing Dependencies

In order to find a solution to the dependency problem I imagined the ideal way to build and use objects. I went back to an early concept in object technology called Software ICs. This was first introduced by Brad Cox [4]. A software IC is like a hardware IC, it has inputs and outputs which are connected to other ICs. If we could build systems by connecting black box object outputs to inputs, we would have a very loosely coupled system which should have the properties I was looking for.

The first decision was to decide what were the black boxes: objects or methods? If a black box was an object, then an input would be a method and an output would be a message send to another object's method. If the black box was a method, then the inputs would be the method parameters and the output would be the method's output parameters. Methods as black boxes is similar to a data flow approach. Morrison[5] has done some interesting work with "Flow Based" programming which uses this approach. However, I was more interested in building and reusing objects, not methods, so I opted for the first approach - using objects as black boxes.

Objects as independent black box components have the following properties:

1) They maintains their own state. A component cannot rely on any other component for state information, as that would create a dependency.

2) A component cannot query another component. When an output of a component is connected to the

input of another, the data flow is strictly in one direction.

3) A component can contain other components, but can only have one parent. This is consistent with the clear boundaries of a black box.

In respect to other component models (COM, CORBA, etc.), the component protocol described here is not intended to be a competing protocol model. Rather, the components described are building blocks used to design and implement software systems. They are more detailed, internal building blocks which could be used to implement an OLE or CORBA component, or to implement stand alone applications.

The API of a component for our purposes has three sections: inputs, outputs, and controls. Component inputs set the state of the component or pass data to the object to process. Outputs of a component are generated when the state of a component changes or as the result of an operation of the component. Controls are used by the parent object to set, query and control the sub component. Since the parent's behavior is implemented by its sub components, parents can have intimate knowledge of the sub components behavior.

Diagrams are used to document a parent's components, and how these components are connected. In the diagram, an object class is represented by a gray box. The input and output ports are displayed inside the class box, and are connected with directed lines. Each line is labeled with the parameters (if any), sent from the output port to the input port. A second list is used in the class box to show the control ports available to the parent.

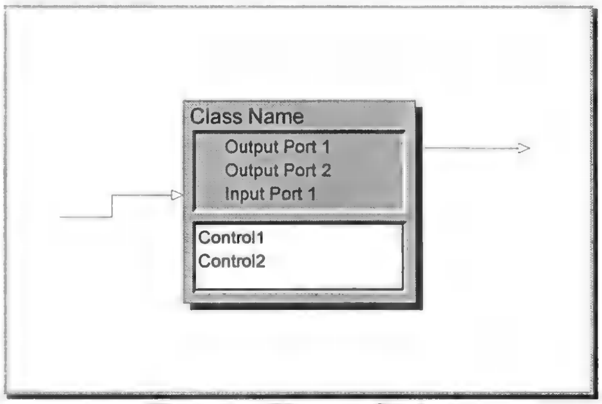


Figure 1 - Diagram Syntax

Building Systems with Components

Given this definition of components, systems are built by parent components connecting the outputs and inputs of its sub components.

A parent can connect sub components for one of three purposes: to implement a sequence of messages, to broadcast a state change, or to pump the output stream of one component to the input of another.

o Sequence of Messages

This usage is similar to a data flow style of programming. Each component transforms its inputs to outputs and sends them to the next component in the chain.

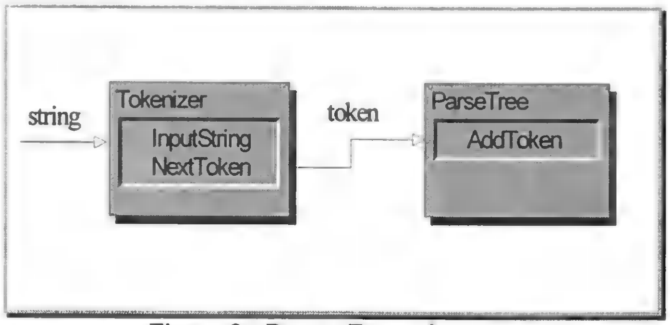


Figure 2 - Parser Example

This example is a parser which must tokenize its input and generate a parse tree. Here, a string is sent to the Tokenizer component's InputString port. It creates a token for the string and sends it out its NextToken output port which gets sent to the ParseTree component's AddToken port.

o Broadcast of a State Change

This usage is similar to the MVC (Model-View-Controller) architecture developed in Smalltalk. A change in a model (or business) object is broadcast to all its views which are then updated.

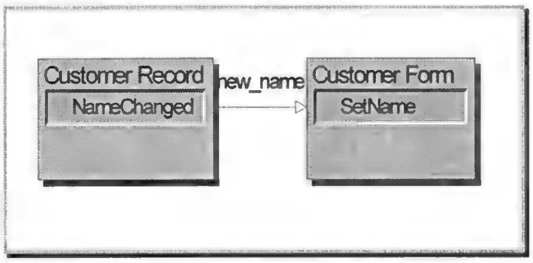


Figure 3 - Customer Entry Example

This example is from an application which lets the user look at and change customer records. Here, when the name is changed in the CustomerRecord component, the NameChanged output is triggered which sends the new name to the CustomerForm, which updates the screen. Note that the CustomerRecord would have a number of output ports for various state changes. Also note that this use is different from a typical MVC setup, because here the "View" object does not have to query the model for the data, the data is always passed. Since the view object doesn't query the model object, it is not dependent on the model object and its protocol. See [6] for more information on MVC.

o Pumping an Output Stream to Another Component

This usage is similar to piping under Unix and covers cases where a set of data is processed by a component and the results are sent to another component.

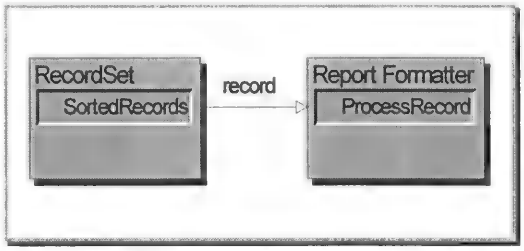


Figure 5 - Report Processing Example

This example is from an application which produces reports given a list of records. Here, the RecordSet component contains a list of records which will be formatted by the ReportFormatter component. The SortedRecords output port pumps all the records in sorted order to the ProcessRecord input port of the

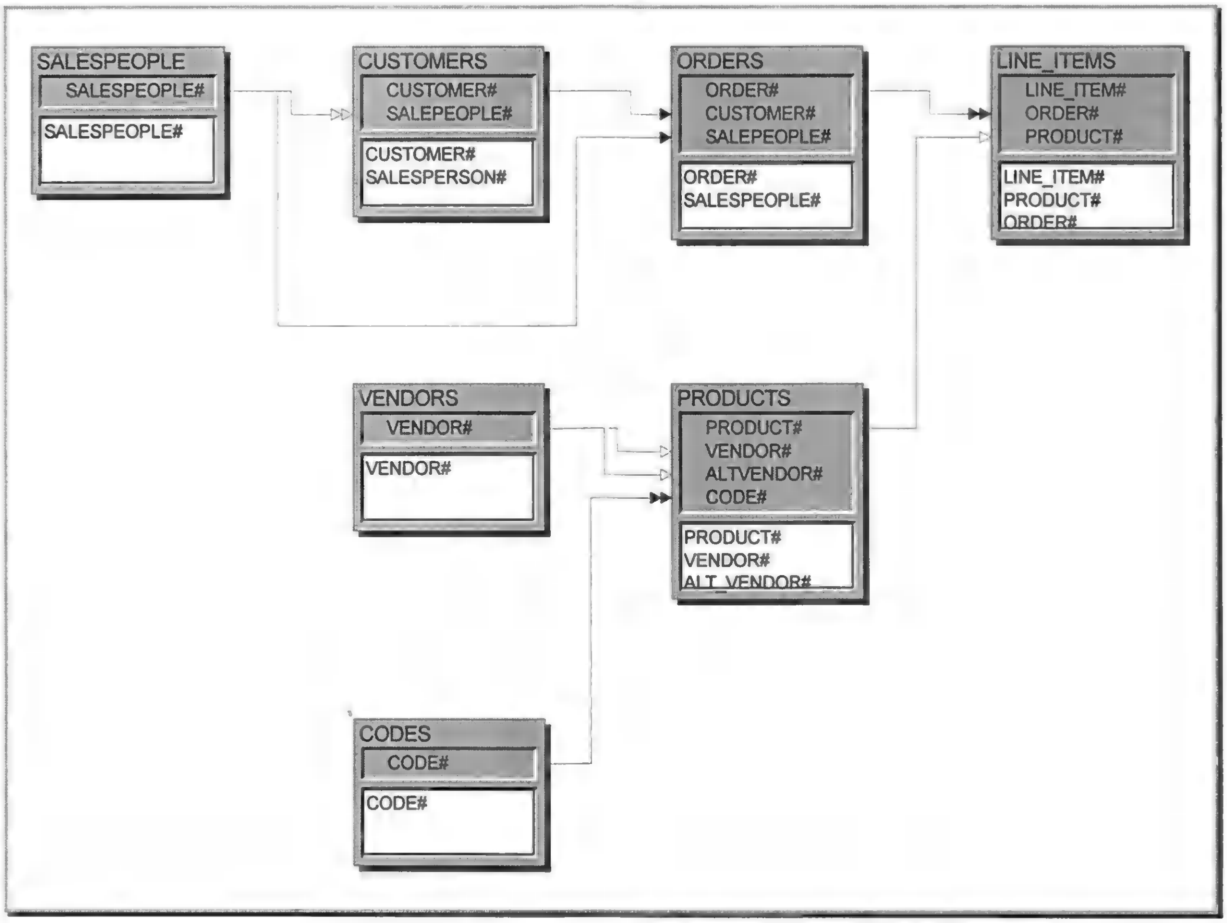


Figure 4 - Sample ER Diagram

ReportFormatter component.

Given these three types of interactions, and the fact that a parent component can control its sub components, the question is: Can we build complex systems with just these constructs? The case study presented in the next section describes a complex system which was successfully implemented with these techniques.

Case Study - an ER Diagram Generator and Editor

The development of Software AG's query and reporting tool "Esperant" was used as a case study for these techniques. Esperant has two components: an administration tool to design the data view, and an end user query tool.

A DataView is a view of the database which is closer to the user's view of the data. Tables and columns can be renamed to more user friendly terms. Tables can be joined to create "denormalized" views of the data , which again is more user friendly.

To define the joins, an Entity Relationship diagram is used. An ER diagram contains table nodes with primary and foreign keys, table columns, and join lines which connect the primary to foreign keys. Figure 5 above shows an ER diagram for our sample database.

The join lines are "locked" to the tables. If a table is moved or resized, the join lines must be stretched, rerouted and possibly clipped to conform to the new position.

Since our Esperant users had existing DataViews, and since joins can be inferred from many database schemas, another requirement was to automatically generate the ER diagram from a list of tables and joins.

Thus the project was to build an ER diagram generator and editor, with standard graphic editing capabilities: selection handles, direct manipulation to move and resize tables and to reroute joins, and undo.

In designing the system using components, it was first decided to separate the automatic generation from the editing. The generation involves computing an optimal layout of the entities and then routing the joins between them.

Thus, the diagram subsystem consists of three major components: a LayoutGenerator, a Router, and a DiagramEditor.

The inputs to the subsystem accept new entities and joins. These inputs are connected to the inputs of all the sub components to update their respective lists (see figure 6).

The LayoutGenerator and Router components produce position information which is sent to the DiagramEditor to position the entity boxes and relationship lines. Note also that the LayoutGenerator sends node position information to the Router, as the router needs to know the absolute entity positions before routing can begin.

In the system, these components work as follows. The "back end" of the system retrieves a list of entities and relationships, either from an existing DataView or by reading the database catalog. The back end is connected to the Diagram SubSystem's AddEntity and AddRelationship input ports. The back end pumps all the new entities and relationships through these ports.

A control message is then sent by the Diagram SubSystems's parent to generate the diagram. The

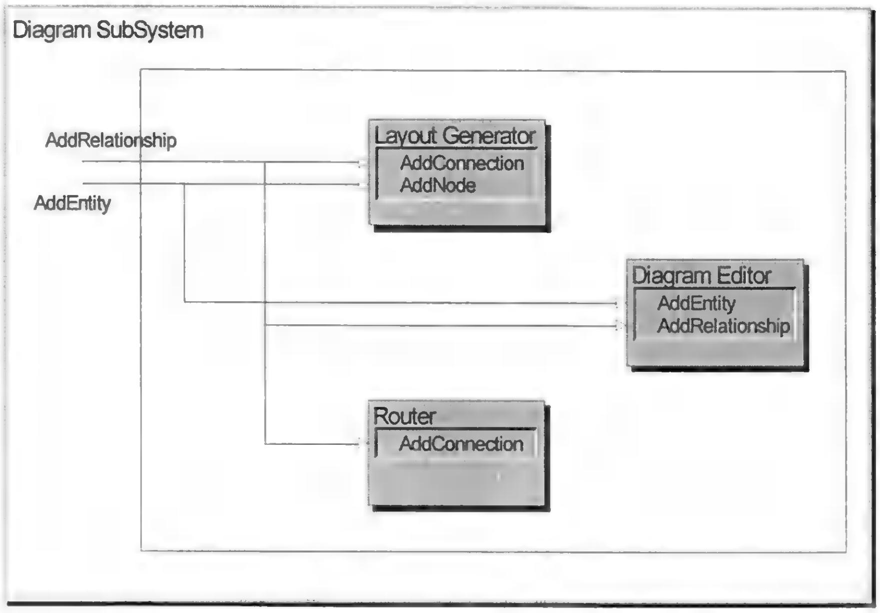


Figure 6 - Diagram Subsystem Components

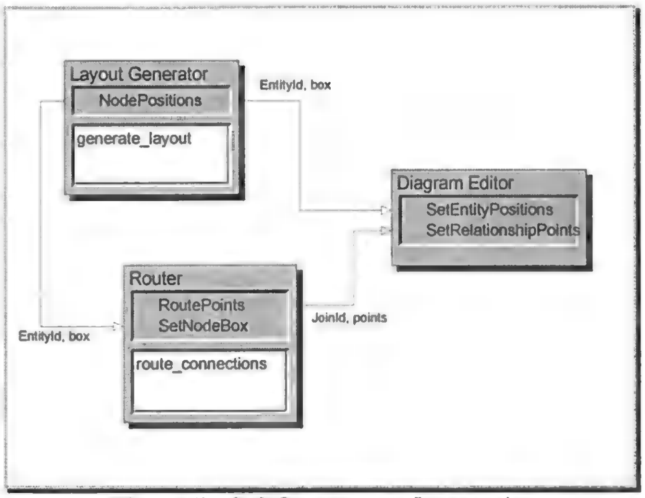


Figure 7 - SubComponent Interaction

Diagram SubSystem triggers the generate_layout control in the LayoutGenerator. The generator computes an optimal table layout and sends the table position information to both the Router and the DiagramEditor. The Diagram SubSystem then triggers the route_connections control in the router, which computes the line routing for all joins and sends the route info to the DiagramEditor (see figure 7).

The generator and router components use specialized algorithms and data structures to compute optimal placements and to generate routes with a minimum number of intersections. These algorithms were not well suited for components and were implemented using traditional objects and methods.

The DiagramEditor was very conducive to sub components. An editor is composed of many objects which are very state related. For example, the relationship lines are dependent on the entity positions; the selection handles for an entity or line are mutually dependent on the entity or line they are manipulating.

DiagramEditor Component

In the ER diagram, whenever an entity is moved or resized all connected lines must be stretched and clipped to the new position.

To model this behavior between the entities and the lines, each line maintains a start_box and an end_box of what it is connected to. The boxes are necessary for the line to intelligently update itself and to clip to the box whenever it is edited. The line has two input ports to update the positions of the start and end boxes. Each entity has an output port to indicate that its box has changed. This is connected to the start_box or end_box input ports of all connected lines. The figure below shows the components and connections used for one such relationship.

Thus when an entity is moved or resized, it sends its new position out the BoxChanged port. All connected lines receive the message and update their start or end box, and reroute and possible clip themselves to align to the new box position.

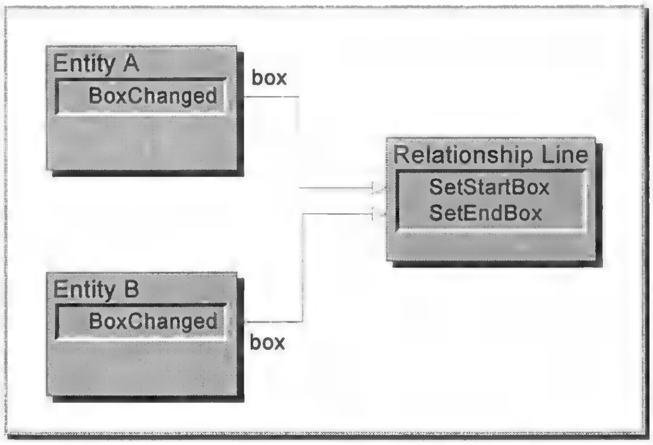


Figure 8 - Entity/Relationship Interaction

Entity and Line Selection

When an entity or line is selected, handles are displayed for the user to move, resize the entity, or reroute the line. The handles are implemented as separate components. They produce outputs when they are edited which are connected to inputs of the entities or lines they represent (see figure 9).

Given the component interactions of the DiagramEditor, whenever an object is changed (moved, resized, rerouted), many objects can be effected (handles update entities which update lines, etc.).

One sticky issue was how to keep track of an invalidation rectangle to redraw when the editing operation was complete. Since an output port cannot query data, one invalidation rectangle could not be built during the message broadcasting. The best solution was to create a separate component which maintains the invalidation rectangle for each transaction.

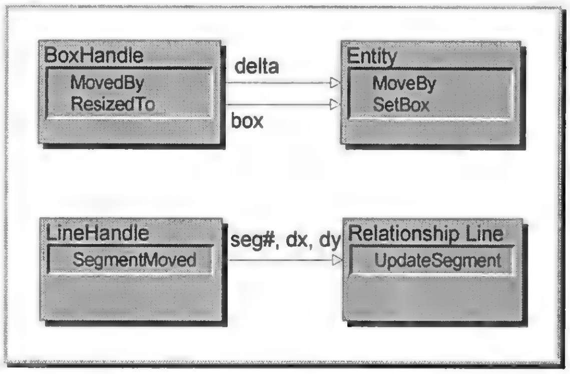


Figure 9 - Handle Interaction

The DiagramEditor contains one InvalidationRectangle component which all entities, lines and handles are connected to (see figure 10). During a transaction, when an object changes its location or size, it generates an InvalRect output message. This is connected to the merge input port of the InvalidationRectangle which collects the rectangles for the transaction. When the transaction completes, the parent DiagramEditor signals the InvalidationRectangle to refresh the window.

A complete edit operation works as follows. The user interacts with a selection handle to drag or resize an object. During the interaction, the mouse messages are routed through the DiagramEditor to the handle or handles effected.

When the user completes the operation by releasing the mouse button, all the effected handles trigger their MovedBy or Resized output ports. These send either a delta or new box to the entity connected to the handle.

The entity updates its position and sends an invalidation rectangle (old position + new position) out its InvalRect port, which goes to the InvalidationRectangle component. It also sends its new box out its BoxChanged output port. This goes to all connected lines which update themselves and send their invalidation rectangles (old bounding box + new bounding box) out their InvalRect port which goes to the InvalidationRectangle component which merges the rectangle with all previous ones.

The DiagramEditor then triggers the InvalidationRectangles's invalidate control which does the actual window system invalidation.

To get a more complete picture of the design of the diagram editor, the above diagrams are combined and shown in figure 11. This represents the primary behavior of the sub components of the editor. In actuality, there are more ports and connections to handle more detailed behavior such as reordering join keys, right mouse menus, drag and drop joining, etc.

These are not shown here for brevity, but the picture gives a very accurate overview of the design of the system.

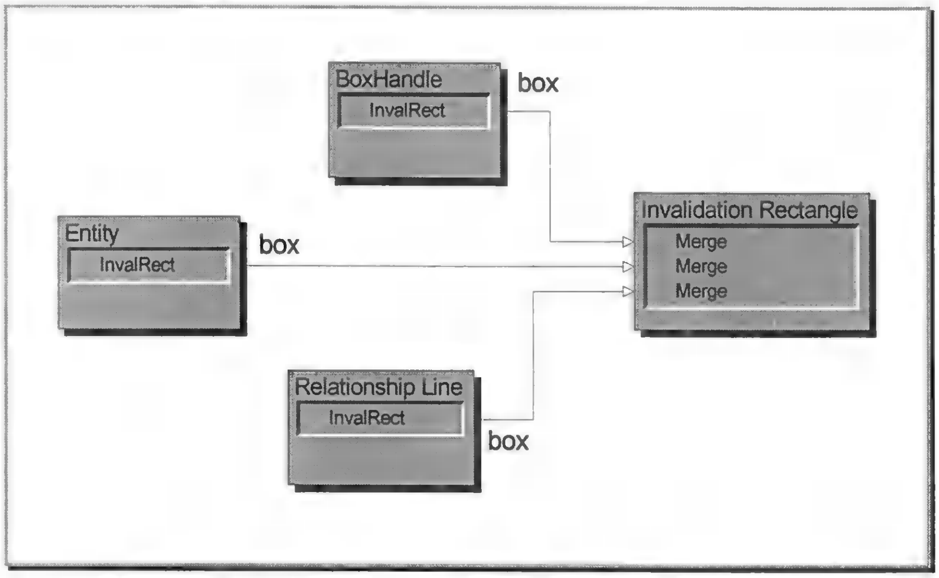


Figure 10 - InvalidationRectangle Interaction

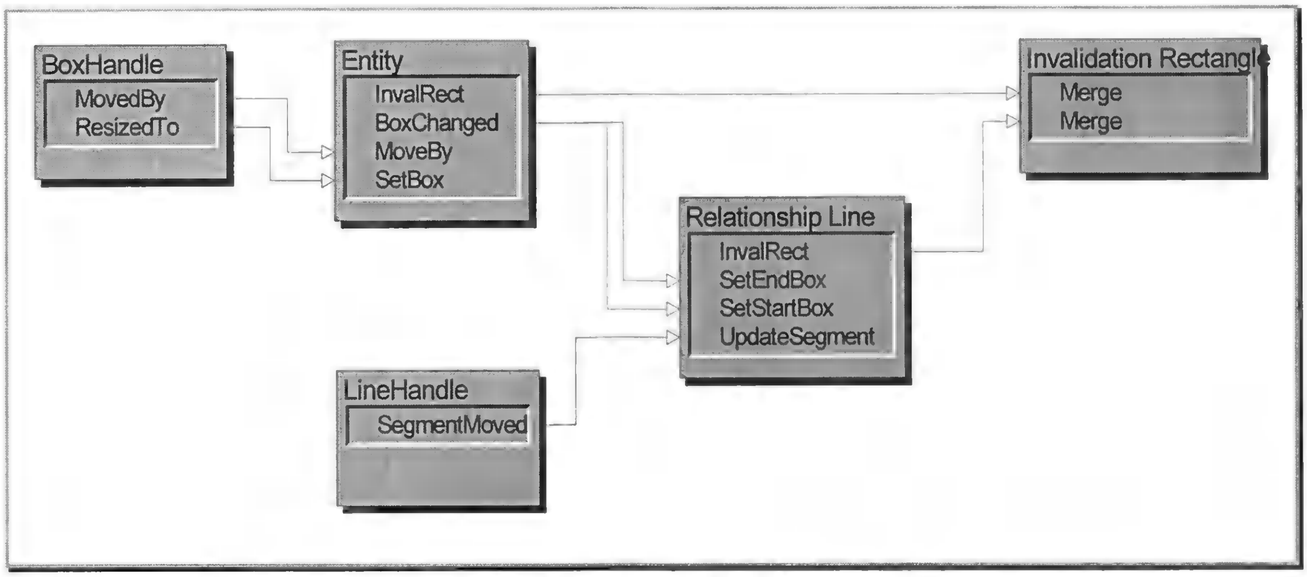


Figure 11 - Overview of Diagram Component Interactions

Implementation of Components in C++

The API of a component has three sections: input ports, output ports and controls. Controls are simple. They are just implemented as methods on the component class.

To connect output ports to input ports, function pointers are used. An input port is a static member function on the component class (note that it must be static, since C++ does not support function pointers to member functions). This static member function calls the appropriate real member function of the component instance.

For example, the input port "MoveBy" on the DrawingNode component is implemented as follows:

```
class DrawingNode
{
static void MoveBy(void *self, int dx, int dy)
  { ((DrawingNode*)self)->_MoveBy(dx,dy);}
void _MoveBy(int dx, int dy);
};
```

The first argument of an input function is the component which receives the message. The recasting is necessary since we cannot use direct member function pointers. The recast is guaranteed to be correct based on the scheme of connecting inputs to outputs (see the following section about type correctness).

Each output port is implemented with a pointer array. We use Visual C++ with MFC and so use a CPtrArray object. The pointer array will contain a list of object/method pairs of all input ports connected to this output. For example, the output port "moved" is defined on the Handle component as follows:

```
class Handle
{
CPtrArray moved;
};
```

Macros are used to connect, disconnect and send data through a port. A connection is established with the macro:

```
Connect(SendingComponent, port, functionType,
                receivingComponent, inputPort)
```

For example, to connect the "moved" output port of a Handle to the "movedBy" input port of a DrawingNode the command would be:

```
Connect(aHandle, Moved, MovedFunc,
        aDrawingNode, MovedBy)
```

where MovedFunc is a typedef which declares the output port parameters and is defined as:

```
typedef void (*MovedFunc)(void *receiver,
                                    int dx, int dy);
```

The connect macro adds the receiver and input function to the Moved output port list. Here, a pointer to the DrawingNode and the input port function "MovedBy" is added to the Moved list of the Handle object.

Data is sent from an output to an input via a Send macro. There are different macros depending on the number of arguments in the message. For example, when the Handle is moved, it will send a message to all its connected components with the macro:

```
Send2(Moved, MovedFunc, dx, dy);
```

This macro iterates through all components connected to the Moved output port and sends the two arguments. The MovedFunc typedef is used to ensure the correct number and type of arguments are used for the port.

Type Correctness

To avoid errors and take advantage of the strong typing of C++, we must ensure when data is sent from an output port to an input port, that the receiver is correct and the arguments are correct. Since connecting the output port to the input port is done in the parent component, and the sending is done in the sub component, the Connect and Send macros must work together to ensure correct types.

This is achieved by using function prototypes for all output ports. By convention, each output port has a corresponding function prototype which declares all parameters of the port. The prototype name is the name of the output port concatenated with the string "Func".

For example, in the API of the Handle component, there is a function prototype called MovedFunc declared as follows:

```
typedef void (*MovedFunc)(void *receiver, int dx, int dy)
```

This function prototype is used by both the Connect and the Send macros to ensure that a) the output port is connected to an input port with compatible arguments, and b) when data is sent out the output port, the data sent is the right type for the port. This is illustrated in the definition of the Connect and Send macros which are defined as follows:

```
#define Connect(sender, outputPort, receiver, inputPort)
{ sender->outputPort.Add(receiver);
  outputPort##Func func = receiver->inputPort;
  sender->outputPort.Add((void*)func); }
```

Note the use of the '##' concatenate operator to create the function prototype name from the output port.

So the example:

```
 Connect(aHandle, Moved, MovedFunc,
        aDrawingNode, MovedBy)
```

expands to:

```
  aHandle->Moved.Add(aDrawingNode);
  MovedFunc func = aDrawingNode->MovedBy;
  aHandle->Moved.Add((void*)func);
```

The assignment to the func variable will cause the compiler to correctly match the parameters or the DrawingNode::MovedBy static member function with the MovedFunc prototype.

The Send macro uses a similar technique for typing and is implemented as follows:

```
#define Send2(outputPort, arg1, arg2)
{  for (int I = 0; I < outputPort.GetSize(); I++)
   {  outputPort##Func func =
                        (outputPort##Func)outputPort[I+1];
      (*func)(outputPort[I], arg1, arg2);}
}
```

So, the example:

```
  Send2(Moved, dx, dy)
```

would expand to:

```
  for (int I = 0; I < Moved.GetSize(); I++)
   {  MovedFunc func = (MovedFunc)Moved[I+1];
      (*func)(Moved[I], dx, dy);}
```

Here, on the jump thru the function pointer (func), the compiler will verify that the arguments specified in the Send macro match those in the function prototype.

Thus we have verified that the input port parameters match the function prototype, and that the send parameters match the function prototype. Therefore we are guaranteed that the input port parameters match the send parameters.

The other issue to address is the recasting of the void* pointer to the component class in the static member function.

In the Connect macro, the static member function pointer is retrieved by using "receiver->InputPort". The compiler will ensure that the static function for the receiver class is used.

By saving the receiver in the output port list and sending the receiver later to the static member function we are in effect doing the equivalent of:

```
receiver->inputPort(receiver);
```

Thus, we are guaranteed that the object instance sent to the class method will always be a "receiver" class object.

Conclusions

The goal of this work was to find a "divide and conquer" strategy to designing object oriented systems, such that pieces could be built and tested independently. By building independent components, we could reduce the exponential growth of complexity as the system grows, and build complex systems by combining smaller, simpler sub systems.

The technique presented was successful for the project described and achieved these goals. All components had very clear boundaries and could be built and tested independently. This is not to say that each component was trivial to implement. The DiagramEditor is a very complex component which manages complex behavior between many subcomponents. By using the connection strategy described, we created the basic architecture for the system. But these connections cannot model all the behavior, therefore much of it is implemented by the parent DiagramEditor. Though the DiagramEditor code can get complex, the good news is that no matter how complex a component is from the inside, this is totally hidden from the outside where it behaves in a well defined way.

Another benefit of the approach was that supporting Undo in the diagram editor was much easier. Since each component is autonomous, each could maintain its own undo stack. The undo feature was then distributed among all the classes and was very straight forward to implement and to debug. Another section of the product did not use components and implemented undo with a Momento pattern (see [7] for the definition of this pattern). The non component approach was more difficult and required more re design and re implementation than the distributed component undo technique.

The component undo strategy where each component is responsible for its own state, is akin to an object serialization strategy where each object reads and writes itself to a persistent store. It is very clean and produces a straight forward implementation which holds up better to change, as it is more localized.

Some of the components used in this project are being reused in other ones. These techniques have made it easier to identify what objects are required to reuse a component (all supertypes of the component and all its sub components). Also, the loose coupling via the input/output ports makes it easy to work a component into other architectures.

Can this approach be applied to all design problems? There are some cases we are trying to apply it to which are difficult. For example, we have a view in our product which is a scrolling list of records. The records are stored in a RecordSet object and are displayed by one or more view objects. How can the view scroll without querying the RecordSet, because querying other objects is not allowed?

We haven't resolved this, but the approach does not have to be used everywhere and can work in conjunction with other techniques. As mentioned in our Diagram Subsystem example, some components were built using standard objects and methods, and had many dependencies within the component. This was shielded from the rest of the system by wrapping these dependent objects inside a component with a clean API.

The real advantage to this approach is that it adds more structure to object design. Classes, methods and inheritance provide structure which has helped in building, understanding and maintaining systems. The additional structure of limiting relationships and defining Inputs, Outputs and Controls for each component does add more restrictions, but these rules provide clear and standard ways to design and document object behavior. To understand a component's external API one has to look at its inputs, outputs and controls. To understand an object's internals, one has to look **only** at that object and all its sub components. This is a tremendous advantage over other approaches which allow object dependencies, and in our experience has produced a more reliable system of reusable parts which is easier to understand, enhance and maintain.

References

[1] Wirfs-Brock, Wilkerson, and Wiener 1990, *Designing Object-Oriented Software*, Prentice Hall

[2] Rumbaugh et al., 1991, *Object-Oriented Modeling and Design,* Prentice Hall

[3] Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. 1992, *Object-Oriented Software Engineering*, Addison-Wesley

[4] Cox, B., 1986, *Object Oriented Programming, An Evolutionary Approach*, Addison Wesley

[5] Morrison, J., 1994, *Flow-Based Programming*, Van Nostrand Reinhold

[6] Lewis, Simon, 1995, *The Art and Science of Smalltalk*, Prentice Hall

[7] Gamma et al., 1995, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley

Interlanguage Object Sharing with SOM

Jennifer Hamilton
C++ Compiler Development
IBM Toronto Laboratory
jenniferh@vnet.ibm.com

Abstract: Object-oriented programming languages may encourage reuse at the source code level, but they inhibit reuse at the binary object level. Differences in object representation make it much more difficult to share objects, even across different implementations of the C++ language, than to share libraries between different procedural languages such as C and Fortran. IBM has addressed this problem through the System Object Model (SOM). The purpose of this paper is to provide a brief description of the SOM and the mapping from SOM to the object models of several languages: C++, Smalltalk, OO COBOL, and to discuss how binary object interoperability can be achieved through SOM.

1. Overview

The IBM System Object Model (SOM) was designed with three major goals: to enable release-to-release binary compatibility (RRBC) of classes, to provide a state-of-the-art object model, and to facilitate inter-languages sharing of objects [10]. While there has been examination of the success of the first two goals, there has been little investigation of SOM's ability to support mixed-language applications and to enable binary object interoperability. Partly this is due to the fact that, until recently, SOM support was only available for the C and C++ languages, which have very similar language models, so interoperability between these languages cannot be considered generally conclusive. Recently, however, SOM support has been introduced for two additional languages: Smalltalk and OO COBOL.

There are two questions to be answered through this paper. The first is whether binary object interoperability is even possible through SOM among such diverse languages as C++, OO COBOL, and Smalltalk. The second, and more interesting, is to examine the feasibility of using DirectToSOM C++ classes (SOM support where classes are defined using the full C++ language syntax) from other languages. This is an important issue because it would provide additional markets for C++ class library vendors who ported their classes to DirectToSOM C++, thereby increasing the set of class libraries available for use from other languages.

2. Introduction

Object-oriented programming languages provide many well-established advantages over conventional procedural programming languages, in particular through support for encapsulation, which groups data with associated methods. However, this grouping also introduces some problems, specifically in the area of release-to-release binary compatibility and interlanguage object sharing. The C++ language, arguably the most commonly used object-oriented programming language, suffers in particular from these problems.

With procedural languages, new versions of library routines can be introduced without impacting existing code, provided that the procedure signatures are kept compatible and new procedure names don't collide with existing client names. While keeping signatures compatible and avoiding name collisions can sometimes be difficult, it is a relatively simple problem compared to that of keeping class definitions compatible in languages such as C++. The problem for C++ is that it is a static language with a large amount of information about the class, such as its instance size, the order and location of methods, and the offset to parent class data, compiled into client code. Thus adding a new data member to a class, even a completely private member, in most cases requires recompilation of client code, including subclasses. In some cases, binary compatibility can be achieved by carefully managing class changes, but migrating a method up the class hierarchy or inserting a new class in the hierarchy always requires recompilation of client code. Languages such as Smalltalk, where class information is managed dynamically rather than statically, do not have this problem.

Object-oriented programming languages also impede the sharing of code between languages. It is relatively easy to call a C library routine from Fortran, or vice-versa, but very difficult, if not impossible, to share objects between languages such as Smalltalk and C++. This is because each language introduces a specific, internal structure for representing object data and associated methods. There is no standard object representation, such as operating system linkage conventions for procedural languages, to enable the sharing of objects across different languages. Even within a programming language, object sharing is not readily achievable. This is a particular problem for C++: there is no standard object representation defined for the language, so each compiler implementer must choose a layout. Unless the layout is identical between two compiler vendors, objects cannot be shared between these implementations.

Object-oriented programming is intended to promote code reuse and allow changes to be made to class implementations without affecting client code. This source level solution leads to a new set of problems with release-to-release binary compatibility and interlanguage object-sharing. As there is much work being done in the area of class libraries and frameworks, it is particularly important to solve this binary object problem so that class library providers can supply updated versions of their classes without forcing recompilation of existing client code. Further, class libraries should be usable from different languages, or at the very least different language implementations, without requiring multiple versions of the library for each target language or implementation.

3. SOM

The *System Object Model* (SOM) was designed to address the two problems introduced by object-oriented programming languages: release-to-release binary compatibility and interlanguage object sharing. SOM provides separation of interface and implementation through a language-independent object model, allowing the class implementation and client programs to be written in different languages. SOM allows a new version of a class to be supplied without requiring recompilation of any unmodified client code. In general, making a change to a SOM class that does not require a source code change in a client, such as adding new methods, instance variables, or even additional base classes, does not require recompilation of that client.

SOM class interfaces are defined using the OMG CORBA (see [2]) standard language called the *Interface Definition Language* (IDL), which is language-independent although loosely based on the C++ language. As an example, the following shows the IDL definition for the SOM class `Hello` with a single method `sayHello`.

```
#include <somobj.idl>

interface Hello : SOMObject
{
    void sayHello();
};
```

The SOM IDL compiler generates *language bindings* for the target client and implementation language corresponding to an IDL class definition. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through simplified syntax that is natural for the particular language. For example, the C++ bindings allow SOM objects to be manipulated through C++ pointers to objects. Currently, the SOM IDL compiler generates bindings for C and C++.

The SOM run time controls the layout and direct manipulation of class instances. All manipulation of SOM objects is performed through standard procedure calls to the run time. The language bindings provide mechanisms to map the native language syntax to SOM run-time calls. As an alternative to defining SOM classes using IDL, several compilers provide *DirectToSOM* (DTS) support, which allows a class to be defined and manipulated completely using the given language, without ever generating IDL. For example, the IBM C++ compilers for OS/2, Windows, AIX, and MVS allow you to define classes in C++, which they then map to SOM classes implicitly.

Figure 1 shows the relationship between the SOM class description mechanisms and the run-time model. Classes are described either using IDL or through native language syntax with a DTS compiler. If using IDL, the SOM compiler generates language bindings for the client and the implementation, and the corresponding language compilers are used to create binaries using the language bindings. No special compiler support is required to process the language bindings. A DTS compiler generates the client and implementation binaries directly. Note that a class client or implementation could be written using a language for which no language binding or DTS support is available ("other client" in Figure 1). SOM objects can be accessed from any language that supports external procedure calls and procedure pointers and that can map IDL types onto the native language types. The client and implementation interact through the SOM run time support. The arrow between the client and the SOM run time is single-ended, representing a one-way relationship, while the arrow from the SOM run time to the class implementation is double-ended because the SOM run time uses the implementation (for allocation, initialization, and destruction of class instances, among other things).

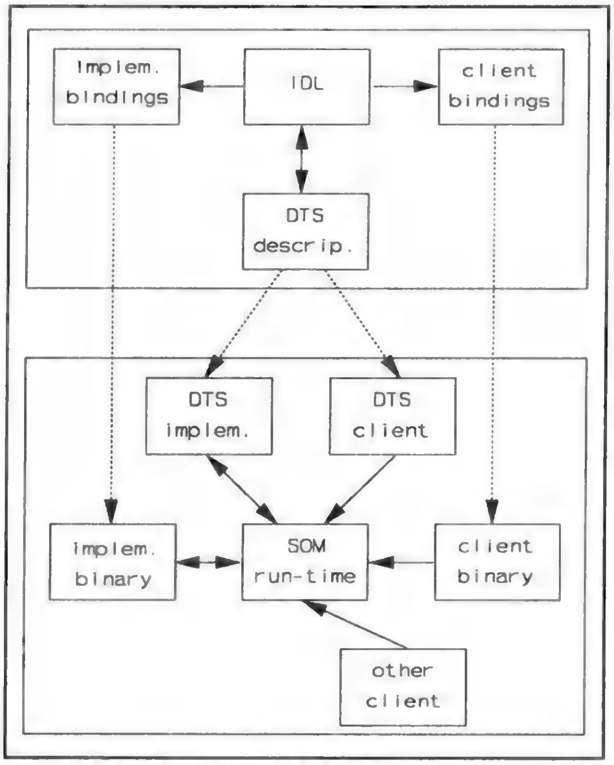


Figure 1 SOM Object Model

3.1 SOM Objects

SOM objects are run-time entities that support a specific interface and have an associated state and implementation. The implementation is only accessible through the SOM object. SOM supports a model similar to that of Smalltalk, in that classes are not purely syntactic entities, as in C++, but are themselves SOM objects. SOM class objects are created at runtime as required by the client, and are used to create and manipulate instances. Class objects support a variety of methods for creating and querying objects, such as determining the size of class instances, whether a method is supported by a given class, and whether a given instance object is a member of that class. A class object is an instance of a special kind of class, called a *metaclass*.

Methods may be invoked on a SOM object in several ways: *offset resolution*, *name-lookup resolution*, and *dispatch-function resolution*. With offset method resolution, the client code invokes the method through a *method token* found at a specific offset in a run-time table. The method token offset is known at compile-time. Name-lookup resolution, by contrast, uses the name of the method to search for the method token. Dispatch-function resolution allows the receiving object to control how the method resolution is performed. Offset resolution is the most efficient means of invoking a method, because the method token is available statically, but the client code is dependent upon the location of that method token not changing. The fixed ordering of the method token table is established by the *release order* for the class.

Every class has a release order, which is simply an ordered list specifying all methods introduced by that class. A client using offset method resolution determines the offset for a method token at compile-time according to that method's location in the release order (which is handled implicitly by language bindings). If a new method is added to the class, at the end of the release order list, it shows up at the end of the method token table, and thus will not impact existing client code. The release order list is the only dependency that a client has upon a corresponding class implementation.

For static clients using offset method resolution to invoke class methods, methods in the release order cannot be removed or reordered without breaking RRBC. New methods can be added only to the end of the release order. Dynamic clients that use name lookup or dispatch-function resolution have no dependencies upon the release order list, and will not be affected if the list is reordered. However, deleting a method from a class could result in a run-time error if that method were later invoked by a dynamic client, becuase that method would not be found.

3.2 Interface Repository

The SOM compiler can optionally create a database, called the SOM *Interface Repository* (IR), which contains class information as supplied by the IDL description. The database can be queried through SOM APIs so that a at run-time a program can access any information available about a class interface. The interface repository content and programming interface conform to those defined by OMG's CORBA Interface Repository. Among other things, the IR provides another mechanism for programming languages to support interaction with SOM. Specifically, Smalltalk language bindings are generated from the IR by the Smalltalk SOM, while OO COBOL uses the interface repository directly, instead of language bindings, to access information about existing SOM class descriptions.

Figure 2 summarizes how the various programming languages that currently provide SOM support access and create class descriptions through languages bindings, IDL, and the interface repository. The next few sections cover the SOM support for C++, OO COBOL, and IBM Smalltalk.

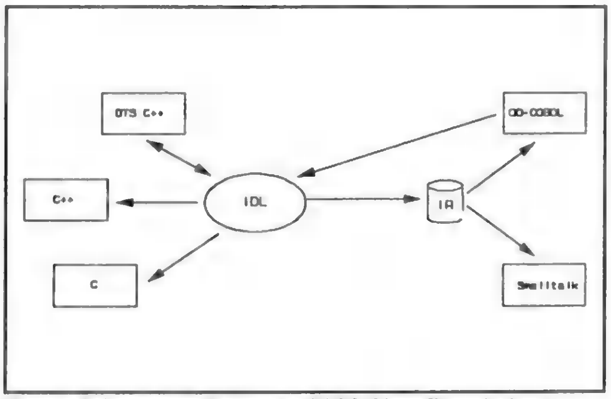


Figure 2 Language Access to SOM Class Descriptions

3.3 DirectToSOM Support

Instead of describing a SOM class using CORBA IDL, DirectToSOM (DTS) support for a programming language allows the class to be described completely in the native implementation language. The compiler generates the appropriate SOM calls and symbols for the class implementation and clients. IDL can be generated from the native language class description if required, or all SOM interactions can be done completely within the native programming language. A subcategory of DirectToSOM, which we call DirectFromSOM, gives client-only capability using native language syntax.

DirectToSOM support is currently supported by two programming languages: C++ and OO COBOL, while DirectFromSOM is supported through IBM Smalltalk. As an example, the code segment below shows a definition for a simple DirectToSOM C++ class. A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file `<som.hh>`. The access specifiers private, protected, and public are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs. The DTS class definition can be used directly by both class client and implementation programs; no IDL description is required.

```
#include <som.hh>

class Hello : SOMObject {
  public:
    void sayHello();
};
```

4. Using SOM with C++

C++ programmers can define SOM classes in one of two ways: either through the C++ language bindings generated from an IDL description, or directly in C++ using a DirectToSOM C++ compiler. The capability to generate C++ bindings from an IDL description allows SOM objects to be created and manipulated with any C++ compiler, gaining the advantages of the RRBC support provided by SOM. In addition, those objects can be shared across different C++ implementations or even with different languages such as Smalltalk. However, in using the C++ bindings, you are limited to a subset of the C++ language, making migration of existing C++ applications more difficult, and you must use two languages (IDL and C++) to define and manipulate objects.

DirectToSOM (DTS) C++ compilers support and enforce both the C++ and the SOM object models, allowing C++ programmers to take advantage of SOM through C++ language syntax and semantics. This makes the use of SOM reasonably transparent and efficient. Instead of first describing SOM classes in IDL, the DTS C++ compiler translates C++ syntax to SOM. You can then have the compiler generate IDL from your C++ declaration, or you may find that you don't need to deal with IDL at all and can work exclusively in DTS C++. And, because you write C++ directly, you can use C++ features in your SOM classes that aren't available through the language bindings, features like templates, operators, constructors with parameters, default parameters, static members, public instance data, and more. The DirectToSOM support is of particular interest in this paper, as it allows existing classes to be migrated to SOM within the confines of the C++ language. Further details about the DirectToSOM C++ support can be found in [4], [5], [6] and [7].

A C++ class is made into a DTS C++ class by inheriting from the class `SOMObject`, which is defined in the header file <som.hh>. You can do this explicitly, as shown above, or implicitly, through compiler switches or pragmas that insert SOMObject as a base class. The access specifiers private, protected, and public are supported for SOM classes and enforced following the C++ rules, as are constructors and destructors and most other C++ constructs. You can create SOM objects statically or dynamically, as simple objects, arrays, or as embedded members of other classes, or anywhere else that the declaration of a C++ object is valid. Most of the C++ rules and syntax apply to DTS classes and objects, with some restrictions. Because the size of a SOM object is not known until run time, compile-time constant expressions such as sizeof are treated as run-time constant expressions. Such operators can still be used with SOM objects, but not in contexts that require compile-time evaluation.

A major inhibitor to RRBC with C++ is the fact that so much information about an object is statically compiled into client code, in particular the location of instance data and virtual function pointers. Data layout and method calling for a DTS C++ class are done using the SOM API, instead of the native C++ API. When you run a program defining a DTS C++ class, the compiler creates the corresponding SOM class object at run time and uses it to create and manipulate the object. As a result, unlike a standard C++ object, much of the information about a SOM object and its class, such as the instance size, is not determined until run time, when the class object is created. This enables class evolution without forcing recompilation of client applications.

C++ instance data members in a DTS class are regrouped into contiguous chunks according to access, in the order of declaration within the class. This regrouping gives efficient access to data members from client code, while enabling RRBC. The location of each chunk is determined at run time. If the declaration order of public and protected data within a class is not changed, and new members are added after any pre-existing members of the same access, this scheme allows new data members to be added without requiring recompilation of any code outside the class (except for friends).

A DTS class also has a default release order. It contains, in the order of declaration, all member functions and static data members introduced by the class, including those with private and protected access. Using the default, you must add any new member functions or static data members at the end of the class. Instead of relying on declaration order, you can instead use the a pragma to specify the release order, in which case you can add new release order elements anywhere in the class, but you must add their names to the end of the list.

For DTS classes, instance data and the release order list are accessed through the SOM run time when manipulating SOM objects, rather than through the statically-defined compiler constructs used by standard C++. This approach provides for both RRBC and an implementation-independent object model. As long as the order of list elements does not change and new elements are added to the end of the list, you can add new data members and member functions without forcing recompilation of client code. In the same way, you can migrate a member function up the class hierarchy. This model solves the fragile base class problem, allowing changes to be made to a base classes without forcing recompilation of derived classes. Further details on the support and restrictions of the model can be found in [4] and [5].

5. Smalltalk

The Smalltalk support for SOM, Smalltalk SOMSupport, is currently available as client-only DirectFromSOM support through the IBM VisualAge for Smalltalk product. The SOM compiler does not create language bindings for Smalltalk from IDL, rather the Smalltalk SOMsupport uses the SOM Interface Repository to create native Smalltalk *wrapper classes* for specific SOM objects. Wrapper classes are generated explicitly through a framework of classes provided with the SOMsupport. The wrapper classes can only be used to create and manipulate SOM objects as a client, there is no support for implementing SOM objects from within Smalltalk, either explicitly, or implicitly by inheriting from one of the wrapper classes. Further details about the constructor and the Smalltalk SOMSupport can be found in [9].

The generated wrapper classes have the same name as the SOM class, prefixed with the string SOM. In order to conform to Smalltalk naming conventions, underscores in class or method names are removed, and capitalization is changed (for example, the first letter of the class name or the letter after an underscore is capitalized). Once the wrapper classes are generated, they become part of the set of classes available to the application. These wrapper classes provide methods that can be invoked using standard Smalltalk syntax, but they map to calls to the SOM API. When an instance of a wrapper class is created, a corresponding SOM class instance is also created. Methods invoked on the wrapper instance result in the invocation of that method in the corresponding SOM instance. As an example of using a SOM class from within Smalltalk, the following shows a Smalltalk code sequence that creates a SOM object of class `Hello`, invokes the method `sayHello`, and deletes the object.

```
| obj |

obj := SOMHello new.
obj sayHello.
obj somFree.
```

A major consideration in mapping from Smalltalk to SOM is memory management. When a Smalltalk object is no longer in use, it is automatically freed by the Smalltalk garbage collector. SOM objects exist outside the Smalltalk memory space, and thus must be destroyed explicitly through the `SOMObject` method `somFree`. Because the absolute memory address of a Smalltalk object can change as the memory manager reallocates storage, Smalltalk cannot pass the actual address of an object as a parameter to a SOM method. Instead, it copies the object to SOM memory and passes the address of the SOM memory copy. However, this copy exists only for the duration of the method call, so SOM implementation methods cannot reliably store this address and use it later. Thus, if the object needs to be referenced in the future from within the implementation, a deep copy of it must be made, rather than simply storing the address.

6. OO COBOL

The proposed ANSI OO COBOL definition [1] extends COBOL-85 with support for object-oriented programming. There are no COBOL bindings for SOM, as there are for C++, which would allow any OO COBOL compiler to access SOM. Instead, the IBM COBOL compilers provide DirectToSOM support by using SOM as the native object model for implementing the OO extensions. Native language syntax is used to define SOM objects, as with DirectToSOM C++, although the use of SOM is much more explicit with OO COBOL. Therefore, for the purposes of this discussion, language interoperability through SOM, the COBOL discussion is restricted to the support provided by IBM OO COBOL.

IBM OO COBOL supports a subset of the proposed ANSI OO COBOL definition. The spirit of the IBM OO COBOL definition is to provide essential OO building blocks through native support, but to take advantage of the existing support in the SOM API wherever possible. To that end, the language description for these extensions is fairly brief. Native language support, based on the proposed ANSI definition, is provided for describing classes and their methods, defining object variables, and method invocation. The object model itself, however, and most other object support, is implemented using SOM; for example, objects are created and destroyed by invoking SOM methods. The following is a brief overview of the native language support. Further details can be found in [8].

6.1 Class Definition

New native language syntax is provided to define a class. All classes must directly or indirectly inherit from SOMObject, which implies that a class must always have at least one direct parent. Following the SOM model, a class is also an object with a metaclass of `SOMClass`, although the class definition may designate a different class as the metaclass. All metaclasses must derive from `SOMClass`. Multiple inheritance is supported, the detailed semantics of inheritance being defined by SOM.

Instance data introduced in a class is accessible only by the methods introduced by that class and is private otherwise. A class cannot access the instance data of a parent class or metaclass. There is no provision for defining class data that is shared across all classes through the class definition itself, although this can be done through a metaclass.

IBM OO COBOL supports both the use and implementation of SOM objects. The compiler uses the SOM Interface Repository to extract information about referenced classes to perform compile-time static type-checking, such as ensuring that a specified object type supports a given method and that the parameters and arguments are compatible. The compiler also supports generation of IDL definitions for SOM classes defined in the COBOL program. Thus a class implemented in C++, for example, can be used by an OO COBOL program, as an OO COBOL class implementation can be used by a Smalltalk program.

6.2 Methods

A class inherits all methods defined by its parent classes. There is no mechanism for hiding a method introduced by a parent class. A class may introduce new methods and override one introduced by a parent class to provide a different implementation. Name overloading by method signature is not supported; all methods introduced by a class must be uniquely named within that class, regardless of case. When a parent method is overridden, the signatures of the overridden and overriding method must be compatible, which, depending upon the parameter type, means that they must be of the same class or one must be a parent class of the other. If the same method name is introduced by two different parents classes, the signatures for those methods must be compatible and that of the leftmost class in the hierarchy is used. This behavior can be modified by overriding the method in the new class and explicitly invoking the desired method by class name. Method binding is performed dynamically using name-lookup resolution.

There is no native support to automatically define CORBA attributes, but this can easily be achieved by introducing the appropriate _get and _set methods.

6.3 Initialization

No native support or syntax is provided for the creation and destruction of objects. Instead, objects are created by invoking the `SOMClass` method `somNew` against a class object and destroyed through the `SOMObject` method `somFree`. Objects can be automatically initialized and deinitialized by overriding the `SOMObject` methods `somInit` and `somUninit`.

To illustrate the OO COBOL support, the following shows a simple example of a class definition, `Hello`, and a client written in OO COBOL. The two programs are compiled separately and then statically linked together.

Definition of OO COBOL Class `Hello`

```
Identification Division.
Class-id. Hello inherits SOMObject.
Environment Division.
Configuration section.
Repository.
    Class SOMObject is "SOMObject"
    Class Hello is "Hello".
Procedure Division.

Identification Division.
Method-id. sayHello.
Procedure Division.
    Display "Hello from COBOL".
End method sayHello.

End class Hello.
```

Cleint of OO COBOL Class `Hello`

```
Identification Division.
Program-id. Client.
Environment Division.
Configuration section.
Repository.
    Class Hello is "Hello".
Data Division.
Working-storage section.
01 obj usage object reference Hello.
Procedure Division.
    Display "Calling somNew".
    Invoke Hello "somNew" returning obj
    Display "Calling sayHello".
    Invoke obj "sayHello".
    Display "Calling somFree".
    Invoke obj "somFree"
    Goback.
End program Client.
```

7. Defining Language-Independent DTS C++ Classes

This section describes the basic considerations for designing DTS C++ classes that can be used from other SOM-enabled languages, and serves partially to explain the coding practices used in the examples that follow.

7.1 Class and Member Names

One of the major considerations with interlanguage sharing of DTS C++ classes is name mangling. Because IDL is case-insensitive and does not support name overloading within a class, C++ member and class names are mangled to provide unique IDL names. In order to accommodate this, C++ names are mangled using a SOM mangling scheme that loosely follows the mangling scheme used by most C++ compilers. In addition, uppercase characters are converted to lowercase by prefixing them with a lowercase z. z_ is used to mean a real lowercase z. So `somSayHelloZz` becomes `somzsayzhellozzz_`.

Within DTS C++, name mangling does not pose a problem; however, the mangled names tend to be fairly long and unreadable, making them unsuitable for use by other languages. There are several pragmas that can be used to affect DTS C++ name mangling for SOM: `SOMNoMangling`, `SOMMethodName`, `SOMDataName`, and `SOMClassName`.

`SOMNoMangling` prevents the name mangling of class member names and can be turned on for a specific class or for a range of classes in the compilation unit. If the class has overloaded member functions, this causes collisions in the generated IDL, in which case the `SOMMethodName` pragma is also required to give specific names to overloaded members. `SOMDataName` can be used to give specific names to class data members. `SOMNoMangling` does not affect class names, which are at the very least mangled to be case-insensitive by translating uper case letters to the lowercase equivalent preceded by lowercase z. Template class names are mangled further to incorporate type information. Therefore, if the class name contains uppercase letters or is a template class, the

SOMClassName pragma should also be used to ensure that the class name is not mangled.

Note that IDL matches overloaded methods by name only, so if a name is not mangled initially, but is mangled later on, this will break binary compatibility because it is equivalent to changing the method name or to removing the method with the old name and adding a method with the new name. Methods cannot be deleted from a SOM class without potentially breaking binary compatibility. The recommended approach if interlanguage sharing is likely to be required is to always use SOMNoMangling and SOMClassName, and to use SOMMethodName when necessary to handle overloaded member names or special names such as operators.

Mangled or not, names should contain only alphabetic characters or digits, and should begin with an alphabetic character. Underscores, while valid for IDL names, may cause problems in languages such as Smalltalk, that remove them from names.

7.2 Data Members

Most other languages do not have the ability to directly access public or protected data members or static data members. The simplest way to allow other languages to access public data members is by making them into CORBA attributes. This can be done using the SOMAttribute pragma. The DTS C++ compiler implicitly generates _get_*member* and _set_*member* methods that return and set the member value respectively. Note that by default the backing data becomes private and all C++ access to the data members outside the class is through the attribute functions. For performance reasons, the backing data can be made public for direct access from DTS C++.

For public static data members, which are shared across all class instances, the recommended approach is to define a metaclass for the given class and make the static data member an attribute of the metaclass.

7.3 Constructors

SOM requires that a class define a default constructor with no arguments. This constructor is mapped by the DirectToSOM C++ compiler to an override of the SOMObject method somDefaultInit. C++ copy constructors override one of four SOMObject copy constructor methods depending upon the method signature (somxxCopyInit, where xx is determined by the source object being const or volatile). Any other C++ constructors will have mangled names, becuase they are not mapped to any SOMObject methods, and should explicitly be given SOM names via the SOMMethodName pragma.

Through both OO COBOL and Smalltalk, somNew can be invoked to create objects. somNew implicitly calls somDefaultInit as part of object creation, which in turn calls the C++ no-argument default constructor. Object creation can also be performed in two steps by first creating an uninitialized object through invoking the SOMClass methods SOMNewNoInit or SOMRenewNoInit against a given class object, and then invoking somDefaultInit against the object.

This two-step mechanism can be used to call copy constructors, such as somDefaultConstCopyInit or other constructors, from OO COBOL or Smalltalk, which don't have language mechanisms to implicitly call these copy constructors. All SOM constructor methods accept three parameters: the target object, an environment, and an initialization control vector. The initialization control vector is used to prevent a class constructor from being called more than once when a class appears multiple times in the inheritance tree, and it should always be supplied as a null pointer.

7.4 Assignment

If an operator= method is not defined for the class, the compiler supplies overrides of the four SOMobject assignment methods (somDefaultxxAssign, where xx is determined by the source object being const or volatile). One of these methods is called when an assignment operator is encountered. If an operator= method is supplied, then it is called when one of the SOM assignment methods is invoked. The SOM assignment methods accept a first parameter that is an assignment control vector, followed by the source object for the assignment. As with the initialization control vector, the assignment control parameter is always passed as a null pointer and is used to prevent a base from being assigned more than once when it appears multiple times in the hierarchy. This support is not available for operator=. So, a SOM assignment method should be defined in preference to an operator= method.

8. Examples

This section provides examples of sharing code with DirectToSOM C++ (IBM VisualAge C++ for OS/2 Version 3.0.), OO COBOL, (IBM VisualAge for COBOL for OS/2 Version 1.1) and Smalltalk (IBM VisualAge for Smalltalk for OS/2 Version 2.0). The first example is a simple DTS C++ class `Hello`, defined below. Note the use of the `SOMNoMangling` and `SOMClassName` pragmas to control IDL name generation. The `SOMIDLPass` pragma is used to add information to the IDL file that cannot be expressed in C++. In this case, it is used to indicate the name of the DLL that contains the current class. The DLL name is used by the SOM API when a class is dynamically loaded at runtime rather than statically bound to the application. SOM classes should be designed for both situations. The corresponding IDL generated by the DirectToSOM C++ compiler is shown following the class definition.

The C++ client program shows how this class is used in C++ using standard C++ syntax, followed by examples of using this class from OO COBOL and Smalltalk respectively. In OO COBOL, the `somNew` method is invoked against the class object `Hello`, which returns an instance of the class `Hello`. Then the `sayHello` message is sent to the object, and finally the object is deleted via `somFree`. The same approach is followed for Smalltalk. Note that the SOM object must be explicitly freed; the Smalltalk garbage collector does not handle SOM objects. The class implementation is shown at end, written using standard C++ syntax.

Definition of DTS C++ Class `Hello`

```
#include <som.hh>

#pragma SOMNOMangling(on)

class Hello : public SOMObject {
    #pragma SOMClassName(*, "Hello")
    #pragma SOMIDLPass(*, "Implementation-End", "dllname = \"hello.dll\";")
  public:
    void sayHello();
};
```

Generated IDL for DTS C++ Class `Hello`

```
#ifndef __hello_idl
#define __hello_idl
/*
 *
 * Generated on Tue May 16 12:08:56 1995
 * Generated from hello.hh
 * Using IBM VisualAge C Set ++ for OS/2, Version 3.00
 */
#include <somobj.idl>
interface Hello;
interface Hello : SOMObject {
    void  sayHello ();
#ifdef __SOMIDL__
    implementation {
        align=0;
        sayHello: public,nonstatic,cxxmap="sayHello()",cxxdecl="void sayHello();";
        somDefaultConstVAssign: public,override;
        somDefaultConstAssign: public,override;
        somDefaultConstVCopyInit: public,override,init;
        somDefaultInit: public,override,init;
        somDestruct: public,override;
        somDefaultCopyInit: public,override;
        somDefaultConstCopyInit: public,override;
        somDefaultVCopyInit: public,override;
        somDefaultAssign: public,override;
        somDefaultVAssign: public,override;
        declarationorder = "sayHello, somDefaultConstVAssign, somDefaultConstAssign,
somDefaultConstVCopyInit, somDefaultInit, somDestruct";
        releaseorder:
                sayHello;
        callstyle = idl;
        dtsclass;
        directinitclasses = "SOMObject";
        cxxmap = "Hello";
        cxxdecl = "class Hello : public virtual SOMObject";
        dllname = "hello.dll";
    };
#endif
};
#endif /* __hello_idl */
```

C++ Client of DTS C++ Class `Hello`

```
#include <iostream.h>
#include "hello.hh"

int main(void)
{
    Hello obj;

    obj.sayHello();
}
```

OO COBOL Client of C++ Class `Hello`

```
        Identification Division.
        Program-id. "Client".
        Environment Division.
        Configuration section.
        Repository.
            Class Hello is "Hello".
        Data Division.
        Working-storage section.
        01 obj usage object reference Hello.
        01 env usage pointer.
        Procedure Division.
            Call "somGetGlobalEnvironment" returning env.
            Display "Calling somNew".
            Invoke Hello "somNew" returning obj
            Display "Calling sayHello".
            Invoke obj "sayHello" using by value env.
            Display "Calling somFree".
            Invoke obj "somFree"
            Goback.
        End program "Client".
```

Smalltalk Client of DTS C++ Class `Hello`

```
tstHello
        "to test the SOMHello class"
        | obj |

        obj := SOMHello new.
        obj sayHello.
        obj somFree.
```

Implementation of DTS C++ Class `Hello`

```
#include <iostream.h>
#include "hello.hh"

void Hello::sayHello()
{
    cout << "Hello from C++" << endl;
}
```

The second example illustrates the use of a range of C++ class methods, such as constructors and assignment operators. The IDL generated by the DirectToSOM C++ compiler is shown following the class definition. The class `GenericString` contains a default constructor (which maps to an override or `SOMObject::somDefaultInit`), a copy constructor (maps to an override of `SOMObject::somDefaultConstCopyInit`), and a third, non-SOM, constructor that is given a SOM name of setInit. The C++ client shows these methods being used implicitly through standard C++ syntax and explicitly by calling somResolveByName, which dynamically retrieves a pointer to a given method. The class GenericString also overrides SOMObject::somDefaultConstAssign, which is called for operator= in the C++ client. Following the C++ client are examples showing how these methods are used from OO COBOL and Smalltalk. For brevity, the C++ implementation is not shown, but it would be written using standard C++ syntax.

This example also illustrates parameter passing and the use of CORBA attributes. When the SOMAttribute is applied to a C++ class data member such as length, the compiler implicitly defines and generates the methods _get_length and _set_length which retrieve and update the value of the member, respectively. This allows C++ data members to be accessed from other languages, as shown in in the OO COBOL and Smalltalk client programs. However, these data members can still be accessed using data member syntax in C++, as shown in the C++ client. Smalltalk generates the get and set methods for attributes in the Smalltalk style of member and member:, which is illustrated in the Smalltalk client program. IDL parameters can be passed as in, out, and inout. In Smalltalk, out and inout parameters are passed as arrays, where the first element in the array contains the resulting value. The methods asOUTParameter and asINOUTParameter can be used to create an array containing one object. OO COBOL has no direct support for attributes, but these methods can be called directly using their method names as shown in the OO COBOL client program. Note that OO COBOL requires that the CORBA environment parameter be passed explicitly to the target method. This parameter is passed implicitly by both DTS C++ and Smalltalk.

Definition of DTS C++ Class GenericString

```
#include <som.hh>

#pragma SOMNoMangling(on)

class GenericString : public SOMObject {
  public:
    long length;
    #pragma SOMAttribute(length, readonly)
    char *data;
    #pragma SOMAttribute(data, readonly)
    GenericString();
    GenericString(const GenericString&);
    GenericString(char *, long = -1);
    #pragma SOMMethodName(GenericString(char *, long), "setInit")
    GenericString& set(char *, long = -1);
    SOMObject *somDefaultConstAssign(somAssignCtrl *, const SOMObject*);
    void clear();
    void display();
    ~GenericString();
  #pragma SOMClassName(*, "GenericString")
  #pragma SOMIDLPass(*, "Implementation-End", "dllname = \"genstr.dll\";")
  #pragma SOMReleaseOrder( \
        length, data, \
        GenericString(char *, long), \
        set(char *, long), \
        clear(), display())
};
```

Generated IDL for DTS C++ Class GenericString

```
#ifndef __genstr_idl
#define __genstr_idl
/*
 *
 * Generated on Fri Jul  7 16:38:24 1995
 * Generated from genstr.hh
 * Using IBM VisualAge C++ for OS/2, Version 3
 */
#include <somobj.idl>
interface GenericString;
interface GenericString;
#include <somobj.idl>
```

```
 interface GenericString : SOMObject {
     readonly attribute long  length;
     readonly attribute string  data;
     void  setInit (inout somInitCtrl ctrl, in string p__arg1, in long p__arg2);
     GenericString  set (in string p__arg1, in long p__arg2);
     void  clear ();
     void  display ();
#ifdef __SOMIDL__
     implementation {
         align=4;
         length:
cxxmap="length",offset=0,align=4,size=4,nonstaticaccessors,private,publicaccessors,cxxdecl="long
length;";
         data:
cxxmap="data",offset=4,align=4,size=4,nonstaticaccessors,private,publicaccessors,cxxdecl="char*
data;";
         somDefaultInit: public,override,init,cxxdecl="GenericString();";
         somDefaultConstCopyInit: public,override,init,cxxdecl="GenericString(const
GenericString&);";
         setInit:
public,nonstatic,init,cxxmap="GenericString(char*,long)",cxxdecl="GenericString(char*,long =
-1);";
         set: public,nonstatic,cxxmap="set(char*,long)",cxxdecl="GenericString& set(char*,long =
-1);";
         somDefaultConstAssign: public,override,cxxdecl="virtual SOMObject*
somDefaultConstAssign(somAssignCtrl*,const SOMObject*);";
         clear: public,nonstatic,cxxmap="clear()",cxxdecl="void clear();";
         display: public,nonstatic,cxxmap="display()",cxxdecl="void display();";
         somDestruct: public,override,cxxdecl="virtual ~GenericString();";
         somDefaultConstVAssign: public,override;
         somDefaultCopyInit: public,override;
         somDefaultAssign: public,override;
         somDefaultVAssign: public,override;
         declarationorder = "length, data, somDefaultInit, somDefaultConstCopyInit, setInit, set,
somDefaultConstAssign, clear, display, somDestruct, somDefaultConstVAssign";
         releaseorder:
                 _get_length,
                 s__P0,
                 _get_data,
                 s__P1,
                 setInit,
                 set,
                 clear,
                 display,
                 length,
                 data;
         callstyle = idl;
         dtsclass;
         directinitclasses = "SOMObject";
         cxxmap = "GenericString";
         cxxdecl = "class GenericString : public virtual SOMObject";
         dllname = "genstr.dll";
     };
#endif
};
#endif /* __genstr_idl */
```

C++ client of DTS C++ Class GenericString

```
#include "genstr.hh"
#include <iostream.h>

int main(void)
{
    cout << "create s1 statically with default constructor" << endl;
    GenericString s1;
    s1.display();

    cout << "set s1 string value" << endl;
    s1.set("string1");
    s1.display();

    cout << "create s2 statically with string constructor" << endl;
    GenericString s2("string2");
    s2.display();
    cout << "create s3 statically with copy constructor from s2" << endl;
    GenericString s3(s2);
    s3.display();

    cout << "clear s3" << endl;
    s3.clear();
    s3.display();

    cout << "assign s1 to s3" << endl;
    s3 = s1;
    s3.display();

    cout << "create obj dynamically with string constructor" << endl;
    GenericString *obj = (GenericString *)GenericString::__ClassObject->somNewNoInit();
    typedef void *(*mpt)(...);
    mpt mp = (mpt)somResolveByName(obj, "setInit");
    mp(obj, somGetGlobalEnvironment(), 0, "obj", 7);
    obj->display();

    cout << "create obj2 dynamically with copy constructor from s2" << endl;
    mp = (mpt)somResolveByName(obj, "somDefaultConstCopyInit");
    GenericString *obj2 = (GenericString *)GenericString::__ClassObject->somNewNoInit();
    mp(obj2, 0, s2);
    obj2->display();

    delete obj;
    delete obj2;
}
```

Smalltalk client of DTS C++ Class GenericString

```
tstString
        "to test the SOMGenericString class"
        | obj1 obj2 obj3 ctrl |

        obj1 := SOMGenericString new.
        obj1 set: 'Smalltalk object 1' pArg2: -1.
        Transcript cr; show: 'After init & set: obj1 data is ''' , obj1 data, ''''.

        ctrl := Array new: 1.
        SOMGenericString somGetInstanceInitMask: ctrl.
        obj2 := SOMGenericString somNewNoInit.
        obj2 somDefaultCopyInit: ctrl fromObj: obj1.
        Transcript cr; show: 'After somDefaultCopyInit obj2 data is ''' , obj2 data, ''''.

        SOMGenericString somGetInstanceAssignmentMask: ctrl.
        obj2 set: 'Assigned from Smalltalk object 2' pArg2: -1.
        obj1 somDefaultConstAssign: ctrl fromObj: obj2.
        Transcript cr; show: 'After somDefaultConstAssign obj1 data is ''' , obj1 data, ''''.

        SOMGenericString somGetInstanceInitMask: ctrl.
        obj3 := SOMGenericString somNewNoInit.
        obj3 setInit: ctrl pArg1: 'Smalltalk object 3'  pArg2: -1.

        obj1 somFree.
        obj2 somFree.
        obj3 somFree.
```

OO COBOL client of DTS C++ Class GenericString

```
        Identification Division.
        Program-id. "Client".
        Environment Division.
        Configuration section.
        Repository.
            Class GenericString is "GenericString".
        Data Division.
        Working-storage section.
        01 env usage pointer.
        01 str1 usage object reference GenericString.
        01 str2 usage object reference GenericString.
        01 str3 usage object reference GenericString.
        01 objp usage object reference GenericString.
        01 txt pic X(25) value "string1 from COBOL".
```

```
 01 len pic s9(9) comp.
 01 p usage pointer.
 01 nullp usage pointer value null.
 01 somNewNoInit pic x(12) value "somNewNoInit".
 01 mp usage procedure-pointer.
 01 b pic s9(9) binary.
 Linkage section.
 01 txt2 pic X(25).
 01 envMaj pic s9(9) usage binary.
 Procedure Division.
     Call "somGetGlobalEnvironment" returning env.
     Set Address of envMaj to env.
     Display "Calling somNew".
     Invoke GenericString "somNew" returning str1
     If envMaj not = 0
         Perform error-handler.

     move length of txt to len.
     set p to address of txt.
     Display "Calling Set".
     Invoke str1 "set" using by value env by value p by value len returning objp.
     Display "Calling _get_data".
     Invoke str1 "_get_data" using by value env returning p.
     Set Address of txt2 to p.
     Display txt2.

     Display "Calling somNewNoInit".
     Invoke GenericString "somNewNoInit" returning str2.
     Display "Calling somDefaultConstCopyInit".
     Invoke str2 "somDefaultConstCopyInit" using by value nullp str1.
     Invoke str2 "display" using by value env.

     Move "assign string" to txt.
     set p to address of txt.
     Invoke str1 "Set" using by value env by value p by value len returning objp.
     Display "Calling somDefaultAssign".
     Invoke str2 "somDefaultAssign" using by value nullp str1 returning objp.
     Invoke str1 "display" using by value env.

     Display "Calling somNewNoInit".
     Invoke GenericString "somNewNoInit" returning str3.

     Move "setInit string" to txt.
     set p to address of txt.
     move 14 to b.
     Display "Calling setInit".
     Invoke str3 "setInit" using by value env nullp p b.
     Display "Calling display".
     Invoke str3 "display" using by value env.

     Display "Calling somFree".
     Invoke str1 "somFree".
     Invoke str2 "somFree".
     Invoke str3 "somFree".
     Goback.

 error-handler.
     call "somExceptionId" using by value env returning p.
     Set Address of txt2 to p.
     Display "major: " envMaj.
     Display "error: " txt2.
     call "somExceptionFree" using by value env.
     Goback.
 End program "Client".
```

9. CORBA and DSOM

The IDL generated by the DirectToSOM C++ compiler is CORBA compliant, and can be used in a non-SOM environment. While additional SOM-specific information is generated, it is guarded by the __SOMIDL__ macro, allowing the files to be used in a non-SOM environment.

While the examples have not been tested in a distributed SOM (DSOM) environment, they should work, as successful testing has been performed with DirectToSOM C++ and DSOM using the IDL generated by the DirectToSOM C++ compiler. Note that additional information can be supplied in the generated IDL for DSOM support using the `SOMIDLDecl` and `SOMIDLPass` pragmas. `SOMIDLDecl` allows the programmer to specify the IDL declaration that should be generated for a given artifact (for example, a type or a member function). This is useful, for example, in specifying the direction of parameters, which default to `inout` for address parameters and `in` otherwise. The `SOMIDLPass` pragma allows arbitrary strings to be generated into the IDL. Further details on the IDL generation performed by the compiler, including the mapping of C++ types to IDL types, and DSOM support with DirectToSOM C++ can be found in [5].

10. Restrictions and Semantic Differences

In general, DirectToSOM C++ supports most of the standard C++ constructs, including templates and pointers to members. This section covers the major restrictions when using DirectToSOM C++ and the semantic differences between standard and DirectToSOM C++ classes.

- class hierarchy:
  A class hierarchy must contain all DirectToSOM or all standard C++ classes; a mixed hierarchy is not supported. In addition, only a single occurrence of each non-virtual base class is allowed within a DirectToSOM hierarchy. `SOMObject` is a special case, as it is implicitly treated as virtual.

- inline member functions
  Inline member functions are currently generated out-of-line by the compiler so that RRBC won't be compromised. For example, if an inline member function accessed a private data member, any client in which that member function were inlined might require recompilation if a new private data member were added.

- `sizeof`
  Because the size of a DirectToSOM C++ class is not known until run time, `sizeof` is a run-time constant expression for DirectToSOM instances and is not allowed in contexts that require compile-time evaluation. For example, given a SOM class `A`, the declaration of an integer initialized to the size of the type `A` is correct because the size of type `A` can validly be determined at run time in this context. But according to the C++ language rules, an array bound must be a positive integral constant expression, so `sizeof(A)` is not valid in supplying the bound of an array because the compiler cannot evaluate the expression until run time. Note that the value returned by `sizeof` is fixed within a given execution.

- `offsetof`
  `offsetof` depends on the data member access, due to the reordering of data members to enable RRBC (see [4] or [5] for details). Public data starts at offset 0, while protected and private together start at offset 0. In addition, the offset is always relative to class that introduces it. `offsetof(base, base_element)` is always equal to `offsetof(derived, base_element)`. Therefore, you cannot use the offset of a data member within a class to get to the beginning of the instance data.

- data member addresses
  The addresses of nonstatic RRBC data members are not contiguous within a class, one of the few areas where DirectToSOM C++ does not conform to the C++ standard. This is necessary because the member is implemented and treated as a reference, so when you take the address of an RRBC instance, you get the value of the reference, which is the address of the underlying hidden structure.

- initializer list
  You cannot use an initializer list to initialize DirectToSOM objects, because DirectToSOM classes always inherit from the `SOMObject` class. (An initializer list cannot be used to initialize an object of a class that has a base class).

- calling through a NULL pointer
  You cannot call a nonvirtual function through a NULL pointer to an RRBC instance, because the method routing must be performed through the SOM API using a valid class instance.

- casting
  You cannot cast a pointer-to-RRBC object to arbitrary storage or an unrelated type (class punning), because the SOM API depends upon the underlying object layout.

- linking
  All static data and member functions must be defined by link time because they are used to construct the class tables.

11. Usage Considerations

Apart from the issues raised above, one of the major considerations when deciding to make a class a DirectToSOM class is performance. While DirectToSOM C++ gives you the power and flexibility of the SOM language-neutral object model, there is an overhead in using it. For interfaces involving many calls to very simple methods, this overhead can be noticeable. However, much ofr the overhead is not intrinsic to the SOM model, and will probably be alleviated in future releases of the product.

In terms of run-time performance, every SOM virtual method invocation currently requires an indirection through a pointer in the SOM class data structures to invoke the method. This is in the form of a call through a function pointer to a *thunk*, which is a small code sequence that performs a few setup instructions and branches to the target method. Nonvirtual and static member functions are also currently called through a function pointer.

In addition, instance data access, either from the client or within the implementation, currently requires a *data token* function pointer call through the SOM class data structures to retrieve addressability to that data. For access through the `this` pointer, one call is required within each method invocation. (For CORBA attributes, this calls is in addition to the `_get/_set` method call, if any is used, because the target method must also access the instance data through the data token. Note that, as mentioned earlier, instance data can be made into attributes to provide external access, but can still be accessed from DirectToSOM C++ directly through the data token without calling the `_get/_set` methods). A future release of DirectToSOM C++ will probably lessen this overhead by accessing instance data for the `this` pointer through a supplied register value. Backend support for specifying that the result of the data token function call as invariant can also improve performance by allowing optimizations such as hoisting data token calls out of loops.

With respect to run-time storage requirements, each SOM object contains a pointer at the beginning to the class run-time data structures. So, when a class is made into a SOM class, each object will increase in size by the size of the pointer.

In certain situations, defining a class as a SOM class can also result in an increase in the static object size. Because inline methods are currently generated out-of-line, the object size may increase noticably over using native C++ for classes with a large number of inline methods. In addition, prolog code is currently generated for each constructor to handle the initialization control vector that prevents multiple base class initialization. For classes with a large number of constructors, this additonal code can be discernable in the resulting object size.

In most cases, however, there will be a certain set of classes that are exposed in the interface, for which language neutrality support is required. Usually, there will be many more internal classes that do not require this additional functionality. Thus DirectToSOM and non-DirectToSOM classes can be mixed as the needs of the application dictate.

As with using C++ instead of C, or a high-level language instead of assembler, using DirectToSOM C++ for language neutrality support involves a tradeoff between programmer productivity and program efficiency. You can choose to manage language neutrality explicitly and probably produce faster code, but this gains will probably be acheived at the cost of language restrictions and increased program management, which can be both tedious and error-prone. Note also that there are additional advantages to using DirectToSOM C++ over just the language neutrality support, such as release-to-release binary compatibility (RRBC) and distributed object support (DSOM). For further information, see [4] and [5].

12. Conclusion

The SOM support for all three languages, C++, Smalltalk, and OO COBOL, is relatively natural and intuitive. In particular, the wrapper classes make the Smalltalk mapping to SOM almost invisible. The SOM model clearly provides a language-independent object model that solves the interlanguage object sharing problem in an almost seamless fashion. Although the examples in this paper are relatively simple in nature, they illustrate that a variety of C++ language features can be used effectively in a mixed language environment. It is therefore quite feasible that a C++ class library could be ported to DTS C++ and used from other languages through SOM.

References

[1] ANSI COBOL-97 Working Paper, December 20, 1994.

[2] *The Common Object Request Broker: Architecture and Specification*. Farmingham, MA: Object Management Group, 1992.

[3] Danforth, S., P. Koenen, and B. Tate, *Objects for OS/2*. New York: Van Nostrand Reinhold, 1994.

[4] Hamilton, J. "Reusing Binary Objects with DirectToSOM C++." *C++ Report*, March 1996.

[5] Hamilton, J. *Programming with DirectToSOM C++*. New York: John Wiley and Sons, to be published in 1996.

[6] Hamilton, J., R. Klarer, M. Mendell, B. Thomson, "Using SOM with C++", *C++ Report*, July/August 1995.

[7] *IBM VisualAge C++ for OS/2 Version 3.0 Programming Guide*. IBM Document S25H-6958, 1995.

[8] *IBM VisualAge for COBOL for OS/2 Version 1.1 Programming Guide*. IBM Document SC26-8423, 1995.

[9] *IBM VisualAge SOMSupport Guide*, 1994. (Online document supplied with IBM VisualAge for Smalltalk for OS/2 Version 2.0).

[10] *SOMObjects Base Toolkit User's Guide Version 2.0*. IBM Document SC23-2680, 1993.

Extending a Traditional OS Using Object-Oriented Techniques

Jose M. Bernabeu-Auban Vlada Matena Yousef A. Khalidi

Sun Microsystems Laboratories

Abstract

This paper describes a new object infrastructure designed for tightly-coupled distributed systems. The infrastructure includes an object model, interface-to-C++ translator, an object request broker (ORB) based on the CORBA architecture model, and a transport layer. The infrastructure allows the use of multiple data marshalling handlers, includes object recovery features, and provides remote communication through a mechanism called xdoors.

The object infrastructure has been used to extend the Solaris™ operating system into a prototype clustered operating system called Solaris MC[1]. This illustrates how the CORBA object model can be used to extend an existing UNIX implementation into a distributed operating system. It also shows the advantages of defining strong kernel component interfaces in the IDL interface definition language. Finally, Solaris MC illustrates how C++ can be used for kernel development, coexisting with previous code.

1. Introduction

Trends in the computer industry indicate that large server systems will tend to be built as distributed memory servers with nodes that are small shared memory multiprocessors. The fundamental reason to build these distributed memory clusters is to obtain scalable throughput and high availability. In addition, by building such systems out of standard commodity nodes, cost-effective systems can be built to serve a particular need.

The model of organization for such distributed systems is still a subject of research. One approach is to view the system as a collection of independent computing nodes that use standard networking protocols for communication. Unfortunately, the operating system in this approach provides little support for building clustered applications and administering the system.

1. Solaris MC is the internal name of a research project at Sun Microsystems Laboratories. More information on the project can be obtained from http://www.sunlabs.com/research/solaris-mc.

At the other extreme, it is possible to take an existing shared memory OS, devised for a shared memory system, and port it to a distributed memory environment. However, it is very difficult to build a highly-available system with such an approach. Also, special support in the networking hardware would be needed to implement cache coherent memory. In addition, performance would be limited unless the operating system was extended to include knowledge of the non-uniform memory access times between nodes.

In between this two extremes there are several possibilities. One of the most promising is to extend an existing OS to work as a distributed OS that doesn't require shared memory. This allows one to build a scalable cost effective multicomputer out of off-the-shelf component nodes, by connecting them through standard high speed interconnects, without the need to have special memory caching hardware. The cluster can then be programmed and managed as a single computer, rather than making the distributed nature of the system available to application programmers.

The Solaris MC project extends the Solaris operating system to multi-computers (clusters) with distributed memory. The main goal is to provide a single-system image of the cluster, while maintaining compatibility with previous application programs. Another requirement is to make the main services of the OS, including the file system and networking, highly available in the presence of failures of one or several nodes.

In such a distributed environment, every service provided by the system uses the communications infrastructure. The adoption of a suitable interprocess communication model, and its associated communications framework was a key design issue in Solaris MC. On the one hand we needed a model allowing us to encapsulate existing kernel components within clean interfaces, and to be able to access them remotely. On the other hand we wanted an efficient communication mechanism that can be tailored to different cluster interconnects. Early in the design phase we made the decision to adopt the CORBA architecture [11], and its associated object model. This led to the need to design an *Object Request Broker* tailored to the needs of a multicomputer OS such as Solaris MC.

Solaris MC borrows some of the distributed techniques developed by the Spring object-oriented OS [10] and migrates them into the Solaris UNIX system. Solaris MC imports from Spring the idea of using a CORBA-like object model [11] as the communication mechanism, the Spring virtual memory and file system architecture [5, 6], and the use of C++ as the implementation language. One can view Solaris MC as a transition from the centralized Solaris operating system toward a more modular and distributed object-oriented OS like Spring.

The rest of the paper is organized as follows. In Section 2 we give an overview of Solaris MC. Section 3 discusses the requirements which gave rise to the particular design of our ORB. In Section 4 we present the main components of the framework. Section 5 describes the handlers, Section 6 explains how extended doors (xdoors) provide communication, and Section 7 gives a sketch of the reference recovery mechanism used by the ORB. Section 8 shows how the framework is used to implement a distributed file system. In Section 9 we describe the status of the implementation. Finally, we conclude the paper in Section 10.

2. Overview of Solaris MC

Solaris MC [7] is a prototype distributed operating system for multi-computers that provides a *single-system image*: a cluster appears to the user and applications as a single computer running the Solaris™ operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same binary and programming interfaces as Solaris, running unmodified applications.

Solaris MC has been designed to run on multi-computer systems formed by independent nodes linked through a high performance interconnect. Thus a Solaris MC system can be seen as a "tightly coupled" distributed system.

The components of Solaris MC are implemented in C++ through a CORBA-compliant object oriented framework with all new services defined by the IDL interface definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts.

Solaris MC builds upon the Solaris kernel by accessing its components through newly defined interfaces. There are, thus, two sets of interface boundaries in Solaris MC:

- New interfaces defined using IDL and built using an object-oriented model. We use a version control scheme based on interface inheritance, and have clearly defined failure modes.
- Existing Solaris interfaces where the implementation of the new IDL-based interfaces plugs into the system. The *vnode* [8] interface is an example of such an interface.

Solaris MC is designed for high availability: if a node fails, the remaining nodes remain operational. Solaris MC has a distributed caching file system with Unix consistency semantics, based on the Spring virtual memory and file system architecture. Process operations are extended across the cluster, including distributed process management and a global /proc file system. The external networks are transparently accessible from any node in the cluster. The prototype is fairly complete—we regularly exercise the system by running multiple copies of an off-the-shelf commercial database system.

2.1 Existing Solaris interfaces

The implementation of the new IDL-based interfaces plugs into Solaris using existing UNIX interfaces. For example, the implementation of the distributed file system interface communicates with the existing system through the vnode interface.

The use of existing Solaris interfaces adheres to the following guidelines:

- The Solaris interfaces are used by the implementation of the IDL interfaces, but new extensions are done at the IDL level.
- Clients of the new extensions only see the IDL interfaces, not the underlying implementations.
- Only (relatively) well-defined Solaris interfaces are used. For example, we use the vnode interface, but we do not touch the internals of the Unix file systems.

The above guidelines are formulated to strike a balance between using the existing system code and achieving our goal of migrating the system towards object-oriented IDL-based interfaces. Figure 1

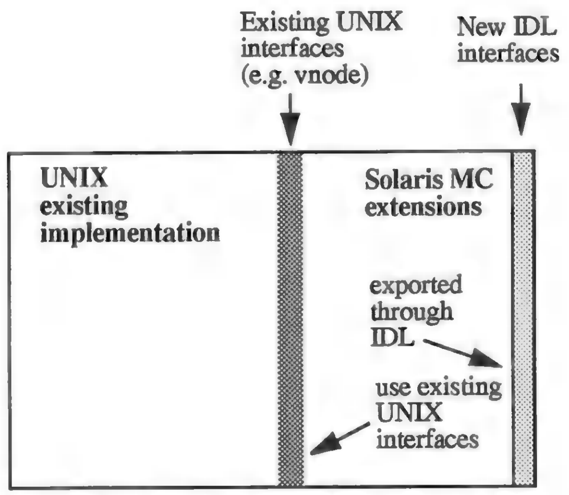

Existing UNIX interfaces (e.g. vnode)
New IDL interfaces
UNIX existing implementation
Solaris MC extensions
exported through IDL
use existing UNIX interfaces

FIGURE 1. New Solaris MC extensions use IDL but build on the existing UNIX system

summarizes the relationship of new Solaris MC extensions to the existing UNIX implementation.

Our intention is to slowly migrate more and more of the existing system interfaces toward IDL.

3. Object Framework Requirements

The object framework underlies the Solaris MC system. In order to build our distributed system, we identified the following framework requirements:

- Support for component encapsulation through strongly-typed interfaces, with well defined invocation semantics and clear failure modes.
- Efficient local and remote object communication.
- Support for building highly available services.
- Ease of integration into UNIX kernel.
- Adaptability to different networking technologies.

Well defined interfaces are essential for long-term system maintenance and evolution. We built well defined interfaces around many parts of the existing kernel in order to take advantage of existing kernel implementations that lacked good interfaces.

The CORBA object model, together with the IDL language, provides sufficient support for defining new components in the system. The CORBA standard also provides a mechanism for remote invocation of the operations in an interface. This is used to communicate between different Solaris MC nodes.

The implementation language must both be able to invoke existing Solaris kernel code, and be suitable for implementing the new code. We use C++ for the implementation language since C++ code can easily call existing kernel code written in C, and C++ code can be easily generated by translation from the IDL specification.

The performance of the communication mechanism selected is an important factor in the performance of the a distributed OS. There are several CORBA-based products on the market right now. However, such products have been designed with user level applications in mind, without considering multicomputer requirements. Thus, it became necessary to design and build an ORB tuned to the needs of Solaris MC.

The communication infrastructure interface is extremely important for good performance, as it critically affects the performance of any inter-node communication mechanism implemented on top of it. In addition, since networking technologies are rapidly evolving it would be an error to design our communication infrastructure around a particular transport.

Finally, in a multicomputer OS, it is not feasible to allow the failure of one component to cause the failure of the whole system. Thus, it is necessary to provide a robust communication infrastructure to allow OS services to withstand system failures.

The above requirements led us to the following decisions:

- Adopt the CORBA object model and the IDL language for interface specification.
- Define an architecture to support the development of highly available services and applications.
- Use C++ as the implementation language.
- Implement our own ORB to provide good performance and integration with UNIX.
- Define a layered structure for the ORB architecture, with well defined interfaces between levels.
- Define a transport facility with both reliable and datagram message delivery facilities. The transport encapsulates all interconnect-specific implementation details.
- Provide support for global reference counting, and automatic notification of the absence of references within the ORB.
- Use a *cluster membership monitor* (CMM) to reconfigure the cluster after a fault is detected. The CMM is used to recover the references possibly lost due to the fault, allowing the ORB to provide strong RPC semantics with clear failure modes.

The result of the above decisions is the Solaris MC object framework.

4. The Solaris MC Object framework

Solaris MC is built using a set of components that extend the base Solaris kernel. The components include most OS services, from file system support to global process management and networking management. The components are implemented as dynamically loadable kernel modules and as user-level servers.

The object infrastructure provides support for implementing the components and the communication between components. The framework includes

- an object model,
- an IDL to C++ compiler,
- an object request broker (ORB), and
- a low-level communication transports.

The rest of this section describes these four components.

4.1 Object Model

Solaris MC components require a mechanism to access each other both locally and remotely, as well as a mechanism to determine when a component is no longer used by the rest of the system. At the same time, each new component must have a clearly specified interface that permits its maintenance and evolution.

Interfaces are defined by using CORBA's *Interface Definition Language* (IDL). Every major component of Solaris MC is defined by one or more IDL-specified object type. All interactions among the components are carried out by issuing requests for the operations defined in each component's interface. Such requests are carried out independently of the location of the object instance by using our the ORB, or runtime. When the invocation is local (within the same address space), it proceeds as a procedure call. When the invocation crosses domains (across address spaces or nodes), the invocation proceeds essentially as an RPC, where the client code uses stubs, and the server (implementation) code uses skeleton code to handle the call.

The CORBA standard also defines some elements outside the basic object model (specifically, the so-called basic object adaptor, a dynamic invocation interface, and a set of object services). We do not provide these features, as we felt that their potential usefulness within Solaris MC did not justify their implementation cost.

4.2 IDL to C++ compiler

The stubs used by the client code and the skeletons used by the server code are generated automatically from the IDL interface definition by a modified version of the Fresco CORBA IDL to C++ compiler [9].

4.3 Object Request Broker

The components of the system communicate with each other using the services of the ORB. The main functions of the Solaris MC ORB are reference counting, marshaling/unmarshaling support, remote request support (RPC), and failure recovery. The two main goals of our ORB architecture are to provide an efficient object invocation mechanism, and support for high availability.

Solaris MC's ORB is composed of three layers: the handler, the xdoor, and the transport (Figure 2). Each object reference is associated with a handler, which is responsible for preparing inter-domain requests to the object whose reference it handles. A handler is also in charge of performing marshaling of its associated references, as well as of local (to an address space) reference counting. The handler layer implements part of the subcontract paradigm [3], providing flexible means of introducing multiple object manipulation mechanisms, such as zero-copy and serverless objects. Section 5 describes object handlers in more detail.

In order to perform an invocation on its object, a handler is associated with one or more *xdoors*. The xdoor layer implements an RPC-like inter-process communication mechanism. This layer extends and builds on the functionality of the Solaris *doors* mechanism.[2] With xdoors, it is possible to perform arbitrary object invocations (intra- or inter-node) following an RPC scheme. References to xdoors are carried out with invocations and replies, permitting the ORB to locate particular instances of implementation objects. Together, the handler and xdoor layers support efficient parameter passing for inter- and intra-node invocations. Xdoors are explained in more detail in Section 6.

The xdoor layer implements a fault-tolerant distributed reference counting mechanism for ORB and application structures. It employs a cluster member-

2. The doors mechanism is a fast inter-process communication mechanism in Solaris 2.5 that is based on the Spring doors IPC [10].

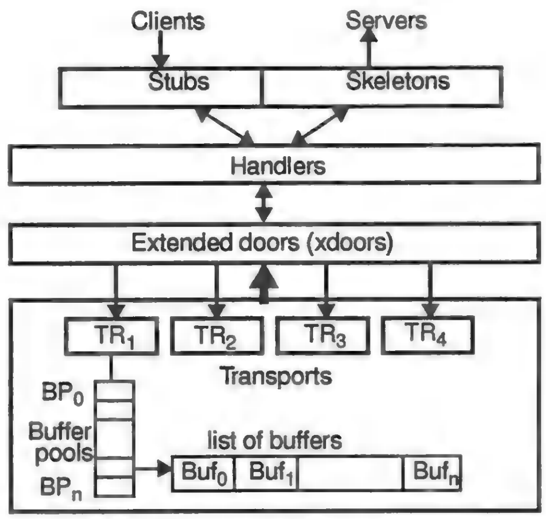


FIGURE 2. Organization of the ORB

ship monitor (CMM) to detect faults and reconfigure the cluster. The same services are also employed to provide clear RPC semantics. In Section 7 we present the general scheme used to recover from faults.

4.4 Communication Transports

The transport layer defines the interface that a communication channel has to satisfy to be used by the xdoor layer. The communication channel would typically be an inter-node high-speed network such as Myrinet [1].

The ORB assumes that the transport implements reliable *sends* of arbitrary length messages. It also assumes the presence of a datagram service, which is used to deliver the CMM messages. The marshaling functions of the ORB request the needed buffers from the particular transport that the request will use. This enables the system to select the best memory to store the marshaled parameters.

To implement efficient parameter passing and zero copy schemes, a transport is capable of gathering information from different buffers to compose an outgoing message. The Solaris MC ORB also assumes that the transport can scatter different pieces of a message into different destination buffers. To do so, the transport offers a set of buffer pools with lists of buffers, waiting to be filled by incoming messages. Arriving messages are stored in the buffers from the buffer pools specified in the message's headers. Since the sender can specify the buffers into which the data should be received, the receiver does not have to copy the received data.

Thus, the transport's interface is both simple and sufficiently powerful to support highly efficient object invocation, providing message delivery with scatter and gather capabilities through the buffer pools.

Together, the handler, xdoor, and transport layers provide support for efficient parameter marshaling and passing. They also make it possible to implement zero-copy schemes for inter-node communications if the communications hardware meets certain minimal requirements.

5. Object Handlers

Each object reference in our framework (whether server or proxy) is associated with an object handler. The functionality of our object handlers is similar to that of *Subcontract* in Spring [2], and *Exchanges* in Fresco [9], providing flexible reference counting and marshaling.

An object handler is involved in all object reference manipulations that are independent of the type (interface) of the object. Thus, the handlers are responsible for keeping accurate reference counting for the objects and delivering an *unreferenced* call to the object when no more references exist. The calls for narrowing, duplicating, and releasing an object reference are all routed through the object reference's handler, which uses them to keep the reference count.

A handler keeps count of the existing references only within the domain (address space) of the handler. The extended doors layer is responsible for keeping count of the number of domains in the distributed system holding a reference to a given extended door. Together, both reference counts keep track of the existing references to a given object (Section 6).

The handler is also involved when marshaling and unmarshaling object references. This makes it possible to create specific handlers for special objects. The handler type to be associated with a particular object implementation is specified when the particular implementation class is defined.

Each handler type comes in two versions, one for the client side and another for the server side. It is straightforward to add new handlers to the system by deriving from the abstract handler class, implementing each of the methods in the interface. The system maintains a small list of system defined handler types. When an object reference is marshaled, the handler type code is included in the marshaling

```
class handler : .... {
        ...
        void unreferenced(ULong count);
        void accept_unreferenced();
        void marshal(service&, CORBA::Object_ptr);
        void invoke(CORBA::Object_ptr,
                        ArgInfo&, ArgValue*);
        void* narrow(CORBA::Object_ptr,
                        CORBA::TypeId,
                        ORB::ProxyCreator);
        void* duplicate(...);
        void  release(CORBA::Object_ptr);
}
```

FIGURE 3. Object Handler interface.

information, making it possible for the receiving node to associate the right type of handler with the object reference it receives. Figure 3 shows part of the object handler interface.

When an inter-domain request is being made on an object, its client-domain handler takes control of the parameter marshaling process through the *invoke* method. Marshaling is accomplished by using helper functions implemented by the ORB, as well as type information generated by the IDL compiler.

A handler marshals a parameter list by writing the marshaled image of each parameter in the *main marshal stream*. The main marshal stream is a logically sequential array of bytes where a domain-independent representation of the parameters is written in the order the parameters are specified in the operation's signature. It is possible to avoid copying large pieces of data by concatenating several buffers at the end of the main marshal stream, forming what we call the *offline buffers*.

To marshal object references, a *special* buffer, separate from the main marshal stream, is allocated. An object reference is usually marshaled in two parts: The first part contains the handler type associated with the object, which is written in its corresponding place within the main marshal stream. The second part contains the internal xdoor number associated with the handler, and is left in the special buffer. The handler for the reference being marshaled determines which additional information is sent within the main marshal stream, the special buffer or as special offline buffers. The only restriction is that the special buffer must contain internal xdoor references only. The above scheme gives the handler writer ample room to implement different marshaling schemes.

In order to release the xdoor associated with a handler when there are no more references to the object within the handler's domain, the handler notifies the associated xdoor that there are no more references. To avoid races, the handler does not go away until the xdoor notifies it with the *accept_unreferenced* method.

On the server side, a similar protocol is carried out from the xdoor to the handler. When the handler decides there are no more references for the object, it invokes the implementation's *unreferenced* method, which is a method on every implementation object. The implementation object can choose what to do in this case.

When the xdoor detects that there are no more remote references, it notifies the handler by calling the unreferenced method. However, the xdoor does not go away until it is also notified through its *accept_unreferenced* method.

Currently we have defined four different types of handlers and are working on a few more types. The types currently defined are the *simple* handler, the *counter* handler, the *standard* handler, and the *bulkio* handler.

The simple handler is used for objects that represent permanent well-known services (e.g., the primary system name service). It does not make sense to keep a reference count for this type of object, so the reference count functionality is left out of the simple handler.

The counter handler is used only on objects which are not going to be exported beyond the limits of a given domain. Usage of this type of handler permits taking advantage of the garbage collection facilities of the Solaris MC ORB even for objects local to an address space.

The standard handler is used for most objects in the system. The handler associates an object with one xdoor, and keeps track of local reference counting, delivering an unreferenced call after the total reference count drops to zero. This handler marshals a reference to its object by simply writing the xdoor number in the xdoor special buffer, followed by the handler type id.

Finally, the bulkio handler implements a zero-copy scheme for page frame lists. The bulkio handler associates a list of page frames with a (pseudo) object reference. When the object is being marshaled in a call to some other interface, the handler marshals the object by attaching an offline buffer containing the list of page frames associated with the object's reference.

The above examples show how the structure of the ORB can be used to implement a variety of schemes for handling object references and parameter passing. Currently we are working on a *high availability* handler that permits a client to transparently switch over to a secondary copy of an object when the primary fails.

6. Extended Doors

6.1 Overview

Extended doors (xdoors) implement the basic RPC mechanism of our framework, relying on transports that provide reliable message delivery. Xdoors also maintain reference counts in the distributed system.

An object can receive an invocation from another domain only if there is an xdoor that refers to it. Each xdoor is associated with just one object (with its handler, to be precise), which may be in the same domain as the door (server xdoor) or may be in another domain (client xdoor).

Xdoors are identified by an integer number within a domain. Client xdoors refer to the location of their server by the server xdoor's local identifier, and the server node identifier. The xdoor layer guarantees the uniqueness of xdoors. That is, any given domain will have at most one client xdoor referring to the same server xdoor.

The xdoor layer maintains reference counts while ensuring the following properties:

- *Liveness*: Once no node other than the server node has a reference to a server xdoor, an unreferenced call is sent to the handler of the server xdoor. The handler itself may decide to send an *unreferenced* call to the object implementation if the total number of local references is also zero.
- *Safety*: No unreferenced call is sent to the handler associated with the server xdoor while there is still a valid client xdoor associated with the server xdoor anywhere in the cluster.

It is important to note that the above properties are maintained even after cluster re-configuration (see Section 7).

6.2 Xdoor reference counting

We implement an optimistic reference counting protocol that fulfills the properties mentioned above without imposing a significant overhead on the system during normal operation. In essence the protocol proceeds as follows:

- When the xdoor layer performs the translation from internal xdoor names to global xdoor names, it increases the external xdoor count of each xdoor whose reference is being included in the message.
- When a domain receives an xdoor reference, if it needs to create a new client xdoor (one that does not yet exist in this domain), it registers the xdoor with its corresponding server xdoor. If the xdoor reference is not new, a de-registration notification is sent right away.
- When a registration is completed, the client xdoor knows that the external xdoor counter of the server has been increased on its account, and can now proceed to send a de-registration notification to the domain that gave it the reference in the first place.
- The sending of a registration or de-registration notification is performed asynchronously.
- When a client xdoor is notified by its handler that there are no more valid references to its object, and its external counter is zero, it sends a de-registration notification to its server xdoor.
- When the external xdoor counter of a server xdoor drops to zero, it calls the unreferenced method of its associated handler. As discussed in the case of the handlers, a special two-phase protocol has to be maintained between the xdoor and its handler to avoid race conditions. Thus the xdoor space will not be reclaimed unless its handler calls it back with an accept_unreferenced method.
- When an xdoor receives a registration notification it increases its external count.
- When an xdoor receives a de-registration notification, it decreases its external xdoor count.

It is straightforward to see that the above protocol satisfies the properties given above for the reference counting protocol.

6.3 Object Invocation Flow

In Figure 5 we show the flow of execution of a standard invocation. The stub's job is to prepare the arguments in a predefined structure which is passed to the handler by calling its *invoke* method. The handler performs parameter marshaling, after which the xdoor is given a buffer list with the invocation data through the xdoor's *request* method. The xdoor function at this point is to get an endpoint from the transport for the node where the server xdoor resides. At the same time it inserts the xdoor reference in the invocation

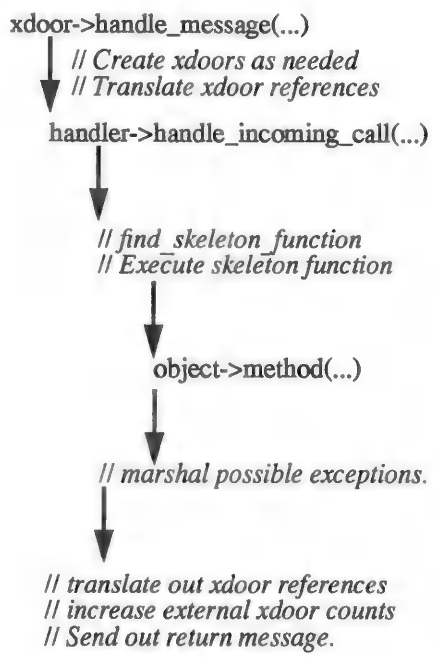

xdoor->handle_message(...)
// Create xdoors as needed
// Translate xdoor references
handler->handle_incoming_call(...)
// find_skeleton_function
// Execute skeleton function
object->method(...)
// marshal possible exceptions.
// translate out xdoor references
// increase external xdoor counts
// Send out return message.

FIGURE 4. Flow of control on the server.

message, and translates the internal xdoor references being passed to cluster-wide xdoor names. When doing so, the external xdoor reference count for the xdoors being passed is increased. After xdoor id translation, the invocation message is ready to be sent. The endpoint's *send* method is used to send the message to the server node.

On return, xdoor references are first extracted from the return message, and client xdoors are created if needed. Then control is returned to the handler, which unmarshals the return parameters. Finally, the end of the stub takes care of preparing the return arguments, and the return value of the object's method call.

On the server side, control flows as shown in Figure 4. First, the invocation message is processed by the xdoor, which inspect the xdoor references being imported, creating client xdoors as needed. At the same time, xdoor references are translated to internal xdoor names.

Afterwards, the handler associated with the server xdoor is called using its *handle_incoming_call* method. This method just finds the skeleton function associated with the type under which the object is being invoked. The skeleton is invoked to perform unmarshaling of the parameters as well as invoking the implementation of the object.

Once the implementation returns, the return parameters are marshaled if no exception was produced. Otherwise, the exception itself is marshaled. At this point, the marshalled return message is ready within the xdoor code. The xdoor translates out the xdoor references being sent back, increasing their external xdoor counts. Finally, the resulting return message is sent back to the client node.

6.4 User-level ORB

The ORB as described so far can be used unchanged either from within the kernel, or as part of ordinary processes. However, this last possibility would force the ORB to consider each process as a "node" in the cluster, with a failure mode independent of every other "cluster" node. In addition, the xdoor layer

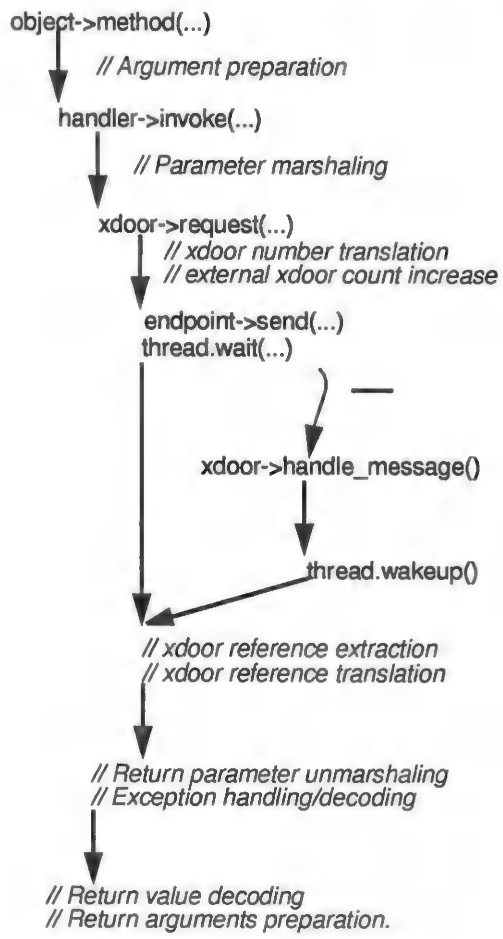

object->method(...)
// Argument preparation
handler->invoke(...)
// Parameter marshaling
xdoor->request(...)
// xdoor number translation
// external xdoor count increase
endpoint->send(...)
thread.wait(...)
xdoor->handle_message()
thread.wakeup()
// xdoor reference extraction
// xdoor reference translation
// Return parameter unmarshaling
// Exception handling/decoding
// Return value decoding
// Return arguments preparation.

FIGURE 5. Execution flow for a standard invocation

cannot reside in untrusted user processes since it can generate a reference to any xdoor in the system.

Therefore, in user-level processes, we use the same ORB down to (and including) the handler level. We substitute the xdoor layer previously described, by a Solaris *door* mediated xdoor layer. A Solaris door is a secure capability-like end point that is used for fast interprocess communication. Thus, a user-level ORB relies on the Solaris door mechanism to perform invocation requests to arbitrary implementation objects, either within the local node, or to remote cluster nodes. Essentially, Solaris doors become a class of xdoors specialized in providing access to user-level implementation objects.

Note that the code is the same for clients, servers, stubs, and handlers when used in the kernel or in user-level processes. This was a conscious design decision, as we wanted to maintain the flexibility of placing a given server or client in the kernel or in user-level processes.

7. Object Framework Recovery

To support high availability, the xdoor layer never aborts an outstanding invocation unless a node is declared to be down. In that case, outstanding invocations to the failed node is aborted, and an exception is raised which can be caught by the handler layer or by the client code itself. To be more precise, an ongoing RPC is only aborted by the system in one of the following cases:

- *Insufficient resources*. The call is known not to have completed.
- *Invalid xdoor*. Such an xdoor may result after cluster reconfiguration, if its target domain is found to be down. The xdoor is tagged invalid, although not immediately deleted. As before, the invocation has not been executed.
- *Node Failure*. The invocation message of the RPC was sent out, and afterwards the cluster got reconfigured, determining that the target node is down. In such a case, the invocation may have been totally or partially executed on the failed node.

An RPC which has been started is never timed out. There are only two ways of returning from an RPC: receiving a reply, or receiving a notification that the other node is faulty. Thus our RPC mechanism implements exactly once semantics if the domain of the invoked object does not fail. In addition, only cluster node failures introduce the possibility of an undetermined execution status for an invocation. Services needing high availability make use of the information in the exception either directly or by means of special handlers (e.g. the high availability handler mentioned earlier on).

The ORB contains a *cluster membership monitor* (CMM) that is capable of detecting faults and reconfiguring the cluster. When the CMM detects a possible failure in the cluster, it runs a membership protocol during the cluster reconfiguration phase. The outcome of such protocol is a group of "surviving" cluster nodes, which form the new cluster.

After each cluster reconfiguration there are two basic elements whose state has to be recovered: the transport and the xdoor layer. Recovery proceeds in phases executed in lock-step by all surviving nodes and controlled by the CMM:

- Initially, the transports are notified of the new group membership. After such notification, transports can stop the send of reliable messages to the failed nodes. Send requests to failed nodes will raise an exception. Reliable messages arriving from failed nodes can be safely discarded.
- Afterwards, ORBs are also notified of the new membership. All ongoing invocations to failed nodes are aborted raising a *node_down* exception on each invocation. At the same time, all client xdoors referring to server side xdoors on failed nodes are flagged as stale. Further invocations through or making use of those xdoors will raise an exception. The ORB can function normally during this phase.
- The third phase recovers the reference counts. Notice that the server xdoor does not keep track of the location of its client xdoors. Thus, after a cluster reconfiguration, the ORB has to proceed through a reference regeneration phase before it can proceed passing references around again. The algorithm followed is optimistic in nature, performing most of its work whenever a fault is detected. Essentially, each node sends every other node its list of client xdoors, such that, node *n* only receives the list of xdoors whose server is in node *n*. All external xdoor counters for client xdoors are set to zero. During this phase, no ORB reference passing activity is allowed.
- Finally, normal operation is reestablished. Server xdoors check their external xdoor count at this moment. Those xdoors having a zero value for that count invoke the unreferenced call of their handler.

If at any point in the above protocol the ORB is notified by the CMM of another reconfiguration, the whole process is started again. At the end, the ORB and transport find themselves in a consistent state with the reference counting system functioning properly. Stale xdoor storage is reclaimed when the handler detects no more references in the domain.

8. Example Subsystem: The Proxy File System

One of the most important components of Solaris MC is its global file system, which makes file accesses location transparent: a process can open a file located anywhere in the system and processes on all nodes can use the same pathname to locate a file. The global file system uses coherency protocols to preserve the UNIX file access semantics even if the file is accessed concurrently from multiple nodes. This file system, called the proxy file system (PXFS), is built on top of the existing Solaris file system at the vnode interface. PXFS interposes on file operations at the vnode interface and forwards them to the vnode layer where the file resides.

The proxy file system makes heavy use of the object infrastructure, exercising most of its features. The files system handles a large number of objects (each file is an object). Because the space overhead introduced by the ORB is small, the impact of the file system on system resources is minimized.

It is very common for different elements in the file system to invoke objects which reside in the same address space. The efficient intra-address-space invocations of the Solaris MC ORB are essential for the performance of the Proxy file system.

Finally, the file system has to move large amounts of data as memory pages. The ability to define new ways to marshal parameters through the definition of special handlers (in particular the bulkio handler), permitted us to implement paging-related operations that avoided superfluous copying of large amounts of data, thus increasing the performance of the file system as a whole.

9. Implementation Status

All functionality described in this paper has been implemented and tested in Solaris 2.5. All Solaris MC extensions are implemented in C++, and are incorporated into the Solaris kernel as loadable modules. In order to mix the C++ code with the existing kernel, it was necessary to create a loadable module containing the C++ runtime support. The basic task of the loadable module is to provide some new relocation types to the kernel dynamic linker. It also supports the "new" and "delete" operators, as well as the C++ exception handling mechanism, which in turn is used to implement our ORB's exceptions.

We make conservative use of C++ features, only using those with sufficient advantages for their cost. Thus, we do not use virtual base classes (the current compiler implementation makes them very space-inefficient), or pass or return objects by value. On the other hand, we find C++ exceptions extremely useful. Exceptions are extensively used throughout our code to signal abnormal conditions and errors.

Preliminary performance numbers are very encouraging. For example, an intra-address space object invocation consists of 9 SPARC instructions vs. 4 instructions for a local procedure call. We plan to report on the performance of the system in a future paper.

10. Conclusions

This paper has illustrated how an object system based on CORBA can be used to extend a traditional monolithic operating system into a distributed cluster operating system. Solaris MC is built as a set of extensions to the base Solaris UNIX® system and provides the same binary and programming interfaces as Solaris, running unmodified applications.

The Solaris MC object system provides several important features. It provides a transport-independent communications layer. It allows multiple object handlers with flexible marshaling and reference counting policies. It uses the xdoors mechanism for reliable RPC. Finally, it includes recovery mechanisms to handle node failures.

The components of Solaris MC are implemented in C++ through a CORBA-compliant object oriented system with all new services defined by the IDL definition language. Objects communicate through a runtime system that borrows from Solaris doors and Spring subcontracts.

Unlike "pure" object-oriented systems such as Choices [2] or Spring [10], Solaris MC is a hybrid system that builds on an existing commercial OS thus leveraging current investment in the OS, device drivers, and most importantly, applications.

Our experience so far with the use of IDL and CORBA to design and implement Solaris MC is very positive. The use of IDL gives us a collection of clearly defined interfaces, and the IDL to C++ translator conveniently creates the glue we need to perform arbitrary service requests from our components. An additional advantage of using IDL is the reduced effort it took us to adapt Spring technology to Solaris MC. The structure of Solaris MC ORB allows us to create special handlers to transmit bulk data and other user-defined structures efficiently between nodes without extra copying.

Perhaps most importantly, the use of the ORB enabled us to provide the system programmers with well-defined and efficient reference counting and failure handling support, that would otherwise have to be programmed, tested, and maintained separately in each subsystem. Building, maintaining, and evolving distributed system is very difficult. By providing our object infrastructure support, we believe we are making the process of building distributed systems easier, less error prone, and in the long run, less expensive.

References

[1] N. J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W. Su, "Myrinet: A Gigabit-per-Second Local Area Network," *IEEE Micro*, February 1995.

[2] R. Campbell and Nayeem Islam, "A Parallel Object-Oriented Operating System." In *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[3] G. Hamilton, M. L. Powell, and J. G. Mitchell, "Subcontract: A flexible Base for Distributed Programming," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[4] Yousef A. Khalidi and Michael N. Nelson, "An Implementation of UNIX on an Object-oriented Operating System," *Proceedings of Winter '93 USENIX Conference*, January 1993.

[5] Yousef A. Khalidi and Michael N. Nelson, "Extensible File Systems in Spring," *Proceedings of the 14th Symposium on Operating Systems Principles (SOSP)*, December 1993.

[6] Yousef A. Khalidi and Michael N. Nelson, "The Spring Virtual Memory System," Sun Microsystems Laboratories Technical Report SMLI-TR-93-09, February 1993.

[7] Yousef A. Khalidi, Jose M. Bernabeu-Auban, Vlada Matena, Ken Shirriff, and Moti Thadani, "Solaris MC: A Multi Computer OS," *Proceedings of 1996 USENIX Conference*, January 1996.

[8] Steven R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of '86 Summer Usenix*, June 1986.

[9] Mark Linton and Douglas Pan, "Interface Translation and Implementation Filtering," *Proceedings of the USENIX C++ Conference*, 1994.

[10] James G. Mitchell, *et al.*, "An Overview of the Spring System," *Proceedings of Compcon Spring 1994*, February 1994.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Revision 1.2, December 1993.

Object Caching in a CORBA Compliant System*

R. Kordale M. Ahamad M. Devarakonda

College of Computing, Georgia Inst. of Technology *{kram, mustaq}@cc.gatech.edu*

IBM T. J. Watson Research Center *mdev@watson.ibm.com*

Abstract

Distributed object systems provide the key to build large scale applications that can execute on a range of platforms. The Common Object Request Broker Architecture (CORBA) specification from OMG attempts to address interoperability and heterogeneity issues that arise in such systems. Our goal is to investigate performance issues for distributed object systems. We claim that object caching is a must for improved performance and scalability in distributed object systems. However, to our knowledge, the CORBA specification does not address object caching. The purpose of this paper is to serve as a data point in a discussion of issues related to caching in CORBA by describing the design and implementation of *Flex*, a scalable and flexible distributed object caching system. Flex is built on top of Fresco that uses the CORBA object model. Fresco runs on the UNIX operating system and our implementation of Flex exploits the features of object technology, Fresco and UNIX. This system allows us to quantify the performance improvements for object invocations that are made possible by caching.

Keywords: Object caching, flexible sharing, open systems, CORBA, UNIX.

*This work was supported in part by NSF grant CDA-9501637 and ARPA contracts N00174-93-K-0105 and DABT-63-95-C-0125.

1 Introduction

Distributed object systems provide the key to build large scale distributed applications that can execute on a range of platforms. Examples of systems which hope to address interoperability and heterogeneity issues include the various CORBA (Common Object Request Broker Architecture) [28] compliant object systems. Existing implementations of these systems experience communication latency inherent in distributed systems as well as the software overheads associated with messages that are exchanged when a client node invokes methods of an object that is implemented by a remote server. These overheads can be reduced significantly if the client node is allowed to cache the object state locally. In this case, future invocations can be executed locally which could lead to better performance.

The benefits of caching are well known and have been demonstrated in systems that range from shared memory systems to distributed file systems. This important technique and the implementation issues related to it have not been addressed in CORBA specifications. A goal of this paper is to contribute towards the discussion about adding caching as a common object service to CORBA. We describe the design and implementation of

Flex[1], a scalable and flexible distributed object caching system. We wanted to build Flex on top of a general purpose and a generally available platform. We chose Fresco [12] which uses the CORBA object model (and was readily available from the X11-R6 distribution). Fresco runs on UNIX and provides an object wrapper around Sun RPC. All invocations on a distributed object reference are translated to remote procedure calls to the object server. In Fresco, object interfaces are specified using OMG's Interface Definition Language (IDL). Flex enhances Fresco by adding caching to it. In addition to the implementation of Flex, issues and solutions discussed in this paper have also been influenced by our previous prototyping of caching in IBM's System Object Model (SOM/DSOM) [15].

Caching has been studied extensively in the realm of file systems [30, 14] and software based implementations [22, 19, 18, 5] of distributed shared memories (DSM) in which files, memory pages, etc. are cached. We will refer to the entities that are cached in these systems as flat objects to distinguish them from the more sophisticated objects (which we will just refer to as objects) which is the focus of this paper. Though object caching has many similarities to caching flat objects, there are a number of important differences.

1. Not all objects encapsulate just data. Many objects are really wrappers around control or hardware devices. Thus, it is not possible or desirable to cache all objects. In particular, it leads to the issue of control over what objects get cached.

2. Objects have references to other objects. Thus, when an object created at a server is cached by a client, it is not always appropriate to cache all objects referenced by the object; some control is needed regarding which of the referenced objects, if any, should be cached. Issues such as this and the previous one leads us to provide object caching as a **Common Object Service** [28] rather than make it a part of the Object Request Broker (ORB).

3. Objects have method level access and it is customary to allow clients to have access only to specific methods of an object. While this is true even of flat objects, they support only two elementary types of accesses - namely, read and write. Security considerations resulting from these differences have implications in the design and implementation of object caching.

4. When methods are invoked on objects, it is harder to detect the type (read or write) of access. This is because objects are accessed using method invocations instead of elementary operations such as read and write.

5. Information about objects is accessed more often than the objects themselves. Object caching, therefore, should also be concerned about caching associated information that is stored outside the object.

While the above design issues are specific to object-oriented systems, some caching issues that are not specific to object-oriented systems can be addressed better by object technology. For instance, caching creates multiple copies of an object which introduces the problem of consistency among the copies. The performance and functionality of a distributed object system could depend very much on the level of consistency it provides. Instead of providing a single consistency level to applications, we take the approach of providing multiple consistency guarantees. In Flex, we take advantage of object technology by allowing class implementors to subclass from a spe-

[1]Our system is not related to the "fast scanner generator" program[31] which is also called Flex.

cific consistency class in the *consistency framework* (described in section 4.1).

The paper is organized as follows. We start with some details on the technique that we use to build multiple consistency levels in section 2. The issues involved in building an object caching system, possible solutions in the context of Fresco and UNIX that take advantage of object technology, and the ones we adopted are discussed in Section 3. Section 4 includes an overview of our implementation of Flex. We present preliminary performance results in Section 5 which demonstrate that caching could result in improved performance for applications that exhibit a reasonable amount of locality of access. Our experience with CORBA and UNIX are detailed in Section 6 and we conclude the paper in Section 7.

2 Building Multiple Consistency Levels

Caching creates multiple copies of an object which introduces the problem of consistency among the copies. The consistency needs differ significantly across different application domains. For example, in a document sharing collaborative environment, changes to documents may not have to be made visible to all users immediately. On the other hand, in a multi-user interactive simulation, changes to the state of shared objects must be made visible to all users immediately and in the same order as the changes were made. Thus, we take the approach of allowing multiple consistency levels to coexist in an application. This can improve performance because of two reasons. Firstly, it allows the use of weaker consistency levels when applicable. Secondly, weaker consistency levels can be implemented more efficiently than stronger ones [23]. In addition to improved performance, a system that provides multiple consistency levels for shared objects can facilitate increased functionality.

Strong consistency cannot be provided in systems where clients are temporarily disconnected which can happen involuntarily or voluntarily in a mobile environment. This is because communication between the nodes, where objects are cached, may be required before an access can be completed when strong consistency is required. Thus, even applications requiring strong consistency can benefit from mechanisms that facilitate a graceful weakening of consistency requirements in order to be able to make progress. We allow applications to employ multiple levels of consistency which can increase functionality and improve performance. In this section, we briefly explain the *mutual consistency* technique that is used to implement multiple consistency levels. The idea behind the technique is to keep cached object copies on a node (machine) mutually consistent.

Informally, copies of two objects are mutually consistent if they could exist together in an application's view. Consider the following collaboration example. Scientist I writes a chapter on results (*results.old*). In this application, each chapter is written as an object. Scientist II reads the copy of results and writes a discussion chapter (*discussion.old*) on the results. Scientist I rewrites the chapter on results (*results.new*) and also a discussion chapter (*discussion.new*) on the new set of results. Now, suppose that a third scientist on a different node tries to read the two chapters. Our definition of mutual consistency specifies that the combination $<$ *results.old, discussion.new* $>$ is not mutually consistent. It is based on the observation that the other three combinations correspond to possible *global system states* [8] that could have occured during the execution of the system. Note that if caching is not employed, only mutually consistent copies of the two objects can be accessed by applications. Of course, the specific two copies that the scientist reads will depend upon the consistency requirements. Thus, a mutually consistent view of a set of objects should correspond to a global system state that is meaningful with respect to the desired consistency level.

An underlying theme of the mutual consistency technique is to ensure that caching overhead on a node is proportional to the amount of caching activity on the node. For example, consider the case in which multiple read-only copies of a strongly consistent object exist when a client wishes to update its object copy. In conventional protocols, this would have induced communication with the other clients that have the read-only copies in order to either invalidate or update them. However, our protocol for strong consistency which is based on the mutual consistency technique does not invalidate or update all the read-only copies. Instead, consistency is maintained by using a novel technique that invalidates some of the locally cached object copies since the idea is to keep the cached copies mutually consistent.

As we saw in the example above, two object copies are *mutual consistent* if the copies and more specifically, their corresponding values, coexisted in a consistent global system state. The more specific question in a test for mutual consistency is whether the "older" object copy is still "valid" in the global system state (view) corresponding to the "newer" object copy. For example, in the above application, *results.old* is not "valid" in the view corresponding to *discussion.new*, since it has been overwritten by *results.new*. Our implementation of the mutual consistency technique uses the notion of a *lifetime* for a value of an object. The lifetime of a certain value of object f is the duration defined by two logical times: the time when this value was created (the *creation time*) and the time until which the system has been able to establish that this value of the object has not been overwritten or become invalid (the *validation time*). Note that the creation and validation times are assigned differently for different consistency levels. For example, suppose that *causal consistency* is desired for an object. Causal consistency only makes use of causal orderings to determine if a cached copy of an object is current. In this case, the creation time assigned to a copy of the object is such that all causally preceding events in the system are reflected. The creation time would be assigned differently if in addition to causal consistency, *coherency* is also desired where coherency requires that all writes to any given object are totally ordered. We call this consistency level as *causal coherency.* In this case, the creation time would reflect causally preceding events as before; additionally, the creation time also needs to be greater than creation times assigned to all previously created copies of the object. Different consistency levels can be implemented using the mutual consistency technique by prescribing rules to assign *creation* and *validation* times to object copies.

The reader is referred to [20] for more details on the mutual consistency technique and the protocols used to ensure causal consistency, causal coherency and strong consistency (SC). SC is related to serializability and sequential consistency which are used in databases and shared memory systems respectively. The current implementation of Flex supports causal and strong consistency.

3 Issues in Object Caching

This section deals with the problems that are independent of consistency levels which need to be addressed in building an object caching system and the various possible solutions.

1. Object Faulting and Access Detection: A method invocation on an object can be executed locally if the object's state currently resides in the cache. Thus, to execute a method invocation, it is necessary to determine if the object state is in the cache. If the object is non-resident, then it must be brought to the client and made available to the program in memory; this is called *object faulting* [13]. Also, concurrent sharing of cached objects leads to the issue of consistency among the copies. Protocols used for maintaining consistency of shared objects may require the system to *detect* updates to object state.

Many present day operating systems allow user-level programs to exploit virtual memory (VM) mechanisms (e.g., *mmap* and *mprotect* calls) to manipulate page protections. These mechanisms can be used for object faulting. VM mechanisms to detect access violations coupled with user defined handlers can be used to implement both object faulting and to detect accesses to shared objects. Several software schemes also exist to facilitate object faulting and access detection that include tagging to distinguish between references to resident and non-resident objects [13]. Object oriented programming languages can exploit the indirection [9] implicit in the method invocation mechanism to fold residency checks into the overhead of method invocation. In IBM's SOM, one could use the BeforeAfter metaclass mechanism [10] which allows a *before* method and an *after* method to be called before and after (respectively) every method of a class. This way, the metaclass can gain control before and after an object invocation that can be used to handle details of object faulting and access detection. In [37], the authors present a method for write detection in a DSM system that relies on the compiler and runtime system.

The VM based scheme induces page faults that have considerable overhead. Also, large page sizes can create problems with false sharing [3]. However, using VM mechanisms induces caching overhead only on object faults and some object accesses. Software schemes, on the other hand, require extra instructions on every fetch and/or store. In [13, 37], the authors argue that software based schemes outperform VM based schemes. However, these arguments are made either in the context of an entry-consistent DSM system [37] or in the context of implementing persistent stores, and garbage collection [13]. On the other hand, a previous study [7] on the issue of granularity choices for data transfer in client-server object-oriented data base management systems (OODBMS) favored page servers (mainly due to the consequent advantages of object clustering). Thus, there is no clear choice and we chose the VM based scheme.

Another issue that arises in object faulting is the difficulty in detecting the type of access - namely, read-only or read-write - while faulting on an object. When a simple memory object is faulted (as in DSM systems), the type of access is readily known since operations are elementary ones (like read or write). However, objects are accessed using method invocations. When a method is invoked on an object, the type of access is not obvious. Faulting an object copy in read-only mode only to realize later during the course of the method invocation that a read-write copy was needed can be inefficient. We take the approach of having the class implementor specify the access type associated with each method invocation. Obviously, such information will have to be placed outside the object in order to decide on the access mode in which to request a copy of the object during object faulting. Consequently, access to such information should be made very efficient possibly by caching relevant parts locally.

2. Control in caching: We alluded to the issue of control in caching in section 1. Here, we recapitulate the issues that lead to the necessity of control. Not all objects encapsulate only data. Some objects are really wrappers around control or hardware devices. Thus, it is not desirable to cache all kinds of objects. Having decided to cache an object, the issue of the most appropriate consistency level for the object remains. Objects have references to other objects. When an object is faulted in, it is not always clear if all referenced objects should also be faulted in. Control over the above aspects of caching can be exercised by one or more of the following entities among others.

1. Class implementor: This is based on the assumption that the class implementor knows best how the object should be implemented.

2. Client application: Usage patterns of objects may vary depending upon the application. In such cases, client application initiated caching and consistency maintenance can be very useful.

3. System: Traditional considerations such as load balancing can be best handled by the underlying system.

Clearly, no single method of control is suitable for all purposes. For example, an implementor of a class may choose to provide caching for objects of that class based on the fact that most applications could benefit from caching such objects. While this benefits applications that need this default behaviour, some applications may have different requirements. For example, they may not choose to cache these objects. Similarly, the system may decide not to cache instances of the class if the machine on which the client application is executing does not have enough resources. Thus, the above scenario illustrates that while the class implementor may be able to decide for the general case, client applications and/or the system may also need control in this regard. In our system, we have implemented class implementor initiated caching.

The class implementor can choose to inherit (directly or indirectly) from one of the classes in the consistency framework depending upon the need for caching and the desired consistency level. Also, one of the classes in the consistency framework (the *Cached* class) provides two methods (*readFromString* and *writeToString*) as part of its interface that provide control to the class implementor regarding the amount of object state that needs to be exchanged between processes. We describe the consistency framework in section 4.1.

3. Cache Organization: Several issues arise related to the memory pool which constitutes the cache. Firstly, clients may or may not be allowed to directly access the cache. For example, in Spring [27], clients do not directly access the cache; instead, they communicate with a local *proxy server* which in turn has direct access to the cache; in other words, objects accessed by clients are actually cached by the proxy server. Such a proxy server provides better security since all client invocations can be intercepted by the proxy server. This design assumes more importance in object caching because it is customary to allow clients to have access to specific methods of an object instead of the entire object. However, every client access to cached objects incurs inter-process communication overhead with this approach. We chose to allow client processes to have direct access to shared objects in the present implementation because of the high overhead of inter-address space communication in UNIX. In the future, we intend to build an user-level RPC (URPC) [4] like mechanism to reduce inter-process communication overhead at which time we will investigate the proxy server approach.

```
class myClass{
  long *ptrVar;
  void set(long *longVar) {ptrVar = longVar;}
  long * get() {return(ptrVar);}
}
```

```
main() {
  myClass *obj;
  ...
  obj = new(...);
  obj->set(new long(10));
  ...
}
        Client A
```

```
main() {
  myClass *obj;
  long *longVar;

  obj = new(...);
  longVar = obj->get();
  ...
}       Client B
```

Figure 1: Per-node caching and legacy applications

Once we decided to allow clients direct access to cached objects, the second issue that arises is whether the memory pool used for caching should be shared across all the clients on a node. We call this scheme *per-node caching.* The alternative is for each client to have its own pool, which we call *per-process caching.*

The per-node caching scheme has the advantage that object faulting and consistency related actions only need to be done for a single copy at a node even when several clients access the cached object at the node. However, there are disadvantages associated with per-node caching. A notable one is the issue of enabling caching in legacy applications. Consider the class definition given for *myClass* in figure 1. Suppose that client applications A and B accessed *obj* (in figure 1) by doing remote invocations on a server and would now like to enable caching for objects of type *myClass.* One way to do it is to define a new class *myNewClass* that inherits from *myClass* and another class that enables caching (such as the *Cached* class shown in figure 2 in the next section). The client application then creates instances of myNewClass. Now, A creates *obj* and caches it. However, when it invokes the *set* method on *obj*, it assigns *ptrVar* in *obj* to an area of memory that is in A's private address space. When B executes, it gets the right handle to *obj* but its attempt to dereference *longVar* results in a segmentation violation.

An obvious solution to this problem is to change the implementation of *myClass* such that the *set* method makes *ptrVar* point to some area in the shared pool of memory that constitutes the cache. One way to do this is to redefine *ptrVar* to be a pointer of type *newLong* and define the constructor of this type to allocate memory from shared memory instead of the process' private address space. However, changing (and recompiling) legacy code may not always be an acceptable option. This issue persists even in other approaches to enable caching such as a scheme that employs runtime inheritance [26].

One way to get around this problem is to route method invocations on shared objects through stub routines that do the necessary marshalling/unmarshalling of arguments. Another approach would be to allow such pointers to private address spaces and handle segmentation faults as they occur. Both approaches introduce additional overhead during each method invocation. In our implementation, we employ *per-process* caching. The reason the scenario in figure 1 does not cause problems in per-process caching is briefly the following. When B tries to access the shared object (*obj*), it cannot get a pointer to A's copy of *obj.* Instead, the entire state of the object is obtained and stored in B's per-process cache thus precluding the problem of accessing memory belonging to another process' address space.

4. Cache Management: As we described before, clients on a node can freely cache in copies of objects to which they have access. To maintain consistency among the copies, mechanisms to invalidate or update object copies and extract state from the object copies are required. Requests for these actions can come asynchronously with the execution of a client process. For example, when an object state is updated at a remote node, the client may receive a message asking it to invalidate its local copy of the object. Such requests can be handled in several ways. However, Fresco's object transport interface is limited by the interface provided by Sun RPC on which it is layered. Moreover, the transport runtime is not multithreaded. Here, we discuss possible solutions in this context.

Our approach is to create a cache manager process which shares the memory pool used for caching with the client and fields external (consistency) requests which may require invalidations or updates and sometimes extraction of the state from an object copy. In our implementation, two processes share memory by opening a common file and *mmap*ing the file to their respective address spaces. We considered three approaches. Problems exist in each of the approaches and we discuss them below:

1. In the first approach, one additional cacher process exists per node which pairwise shares memory with each of the clients and we call this process the *node cacher*. Since the node cacher shares memory with all clients on the node, it needs to open one file per client and may exceed the limit on the maximum number of file descriptors that can be opened by a process[2]. This problem can be solved by managing the open file descriptors independent of the number of client processes on the node.

2. The second approach is to have one additional cacher process per client process and we call such processes *process cachers*. Each client process shares memory with its process cacher. This approach has two problems:

   (a) By doubling the number of processes on a node (one additional process-cacher per client), general degradation in performance can be expected.

   (b) Consider the case where multiple client processes on the same node (say A) have copies of some object. Suppose that a client on another node (say B) wants to access the object and the consistency actions need communication with all the copies (e.g., their invalidations). In order to communicate with the copies on node A, multiple remote invocations from B to A are required. This can be avoided if incoming requests to a node are channelled through some process (for example, the node cacher). This is the approach taken in next case.

3. This approach is the same as the previous one except that it additionally has a node cacher (which does not share memory with clients); all incoming requests are channelled through the node cacher which then contacts the individual process cachers. This approach has the following problems:

   (a) It has the same problem as in case (2a) above.

   (b) This approach incurs the overhead of an additional inter-process communication (IPC) between the node cacher and the process cachers.

The prototype presented in this paper adopts the last approach. However, based on our experiences, we would like to move

[2]Using shared memory segments to implement shared memory between two processes (instead of the mmap approach) has a similar problem.

away from this approach in the next revision of the implementation of the caching system. In particular, we would like to adopt the second approach while moving the node cacher functionality to the object manager or a server.

5. Object Implementation: So far, we have focussed on the issue of managing object state when objects are cached. A related issue that is important in object caching is that of installing, finding, and using code that implements an object's methods; we will refer to the code as the object's implementation. A number of choices exist for dealing with each of the above activities. For example in Emerald [17], object implementations are stored in *concrete type* objects. When a kernel receives an object's state, it determines whether a copy of the concrete type object implementing that object already exists locally; if it does not, the kernel obtains a copy from another node using some location algorithm. In [34], code mobility in Emerald is achieved on heterogeneous computers at the native code level; migrated code runs at native code speed before and after migration.

In the Common ORB Architecture document (CORBA 1.2) [28], on the other hand, it is stated that the object implementation information is provided at installation time and is stored in the Implementation Repository for use during request delivery. Object adaptors in CORBA are responsible for the interpretation of object references and mapping object references to the corresponding object implementations amongst other duties. These functions are performed with the cooperation of the ORB Core and the implementation skeletons. When a client invokes a method on an object, the ORB Core, object adaptor, and the implementation skeleton at the server end arrange that a call is made to the appropriate method of the implementation. A similar mechanism can be used at the client end to use implementations of cached objects, possibly with the aid of a library object adaptor linked with clients that cache objects. This may necessitate that object implementations be installed at each of the sites of use.

This is an important issue particularly in the context of CORBA compliant systems, since an important goal of CORBA is to provide interoperability between applications on different machines in heterogeneous distributed environments and to seamlessly interconnect multiple object systems [29]. In our implementation, however, we make the assumption that the implementation for the cached object is available; in fact, the application code is either compiled or dynamically linked with the code that implements the shared objects.

4 Implementation

The universe of objects in Flex can be divided into a set of inherently private and shared objects. An object may contain any number of references to other objects. The shared objects can be either passive or active. Active objects are associated with a dedicated process usually known as an object server. Active objects are usually large grained and are not cached in our system. Passive objects do not have a process dedicated to them and can be cached.

The system architecture of Flex is defined by the *caching framework* shown in figure 2 which consists of the important classes that enable caching. As shown in the figure, the framework consists of three subtrees. The left-most subtree shows the various styles of accessing distributed objects and this is the framework that is visible to the users. These classes are distinguished with double lines in

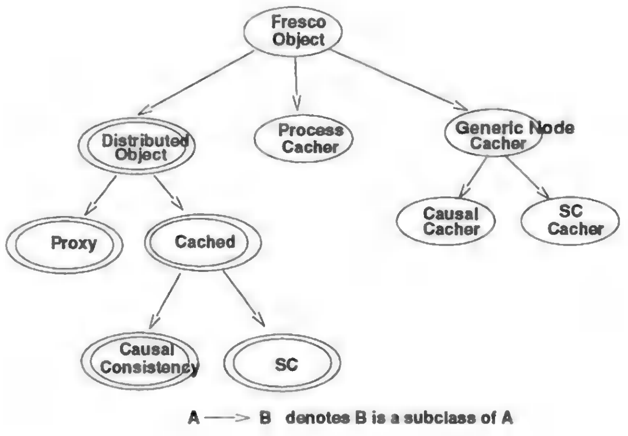


Figure 2: Caching Framework

the figure. Class implementors subclass from one of these classes directly or indirectly while invoking distributed objects. These classes constitute what we call the *consistency framework*. The remaining branches define the classes which are transparent to the application programmers. These define the code that is executed by the various cacher processes discussed in the previous section. These classes define what we call the *implementation framework*.

4.1 Consistency Framework

At the root of the consistency framework is the *Distributed Object* class. Because an object reference is opaque, it is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. The *Distributed Object* class interface provides methods such as *stringify* and *objectify* to solve the above problem. Actually, the object request broker (ORB) interface in the CORBA specification provides more "heavyweight" methods such as the object_to_string and string_to_object to solve a similar problem; these methods in the ORB interface additionally help to convert an ORB dependent distributed object reference to an ORB independent object reference and vice-versa. We provide these methods in the *Distributed Object* class because the implementation of Fresco, that we use, does not support these methods of the ORB interface.

One of the subclasses of the *Distributed Object* class is the *Proxy* class. This reflects the style of access that is specified by CORBA and is generally available in existing CORBA compliant systems. Basically, when a client tries to access an object server, it receives a handle to a proxy object [33]. Any invocation on the proxy object is translated into an invocation at the remote object server. Thus, objects that are not intended to be cached such as wrappers around hardware devices are designed to be instances of classes that are descendants of the *Proxy* class.

Another subclass of the *Distributed Object* class is the *Cached* class. As we shall see later, one of the enabling mechanisms we use to implement caching is to stringify object state, pass it over the network and reconstitute the object at the other end. The *writeToString* and the *readFromString* methods accomplish this. In *writeToString*, the relevant object state is linearized and written out as a string. This string can now be sent out on the wire and the *readFromString* method on the other end can read from the string and update the object's state. Note that this is not the same as the methods we talked above in the context of the *Distributed Object* class. In that class, only the means to access the remote object can be stringified; in contrast, the methods described here stringify the object state and make it possible to transfer object state between processes. A similar interface is part of CORBA's **Externalization** service proposal. Our intention is for the class implementors (who wish that instances be cached and thus inherit from one of the descendants of the *Cached* class) to have the option to override the *readFromString* and the *writeToString* methods defined by default in the *Cached* class. The advantage of such a provision is that the class implementor knows a great deal about the class and therefore can provide optimizations. For example, if instances of a cacheable object have a large state and it is known that the relevant state information that needs to be passed amongst object copies is only a small part of the entire state (e.g., only the modifiable parts of the state), then it is more efficient to stringify only that part of the state instead of the entire object state. Also, as we pointed out in section 3, objects have references to other objects. These two methods allow the class implementor to decide which of the referenced objects need to be faulted in and which of them will just be sent as object references. Of course, if the class implementor does not override these methods, the default mechanism is for a preprocessor to generate these methods automatically.

The subclasses of the *Cached* class implement the specifics related to providing different consistency guarantees. The caching framework shown in figure 2, shows only two subclasses of the *Cached* class that provide strong and causal consistency guarantees. Ultimately, we intend to provide a bigger suite of consistency classes somewhat similar in flavor to the various session guarantees provided in Bayou[35]. The discussion on the relevance of the different consistency guarantees and their details can be found in [20].

4.2 Implementation Framework

The implementation framework can be explained by relating it to the cacher processes that are employed at a node to enable caching (see figure 3). In Section 3, we motivated node and process cachers. While we discussed one node cacher per node, we show three cachers

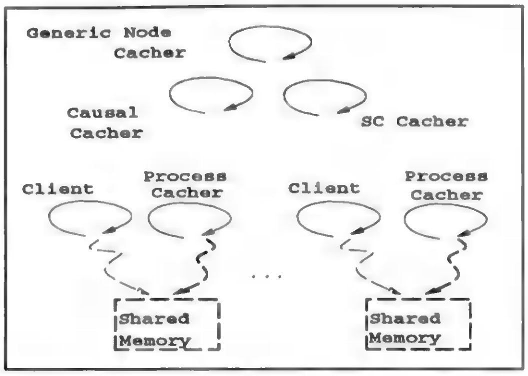


Figure 3: Cacher Processes at a node

at the node level in figure 3 - namely, the *generic node cacher*, the *causal cacher* and the *SC cacher*. We call the causal and the SC cachers as the *consistency cachers*. We chose to use separate cacher processes for different levels of consistency instead of having a single process (the node cacher) handle all the responsibilities of caching at the node level since this design allows new consistency levels to be added more easily. Though we describe a design that allows consistency level of objects to be decided statically, the goal of this prototype is to allow consistency levels of objects to be changed dynamically. This will be facilitated by the generic node cacher which is designed to implement the type-independent aspects of caching. Thus, the generic node cacher class implements functionality that is both type-independent and common to all the consistency cacher classes.

The process cacher class implements the code that is executed by process cachers. A process cacher handles the type-independent aspects of caching that include actions such as maintaining a table of objects that are cached and their corresponding locations in the cache, invalidating the object copy, and invoking methods on the object reference to extract and restore state. The type-specific aspects of object access are implemented partially in the consistency framework and by the specific class that the object is an instance of. For example, if an object fault is experienced while trying to access a causally consistent object, the fault handler that is run is part of the *Causal* class which comes from the consistency framework. Also, as mentioned in Section 4.1, the state of the object is extracted and restored using the default *writeToString* and the *readFromString* methods of the *Cached* class or the overridden definitions of these methods defined in a class outside the consistency framework.

4.3 System API

Object caching does result in some changes that need to be exposed to the application level. These changes affect the way objects are programmed, created and used. First, we allow class implementors to specify the desired level of consistency by having the user defined class explicitly inherit (directly or indirectly) from the appropriate class in the consistency framework shown as part of the caching framework in figure 2.

The second aspect of the API that affects the application program is the following. Since memory pointers do not make sense across address spaces, there needs to be an address-space independent way of accessing a shared object. We call this an *object-id* and it consists of a host name, a process id, and a process-unique identifier. For example, if the object-id of an object is $< hname, pid, nid >$, then this object was created by process with *pid* as its process-id on a host named *hname* and *nid* uniquely identifies the object within the process. A client creates a distributed object on a particular node by specifying the hostname and the process-id while specifying the third component to be zero. When the object is created, its process-unique id is assigned by Flex. Thus, the constructor for the object's class takes an additional argument - the object-id. For example, a client obtains a reference to an existing object by executing **new** className (*oid*, <arguments>) instead of **new** className (<arguments>), where *oid* is the object-id of the shared object that the client wishes to access.

5 Performance

This section presents measurements of execution times of simple invocations on the basic RPC system and also on top of Flex. The experiments were done on two 60MHz Sun SPARCstation 20s running SunOS 4.1.3 and Fresco version 0.7. The machines reside on two different 10Mbps ethernet based subnets connected by a router. The execution times for a single null RPC and a null invocation on a proxy are shown in table 1. The set up in the Sun RPC case is straightforward. A server is started on a node and a client is started on a different node. The client then makes null RPC calls. The set up in the proxy case is also very similar. An object server[3] is started on a node. Clients then specify which object server they need to talk to in a semi-transparent manner and obtain a reference to the object; this is really a reference to the object server. The client then makes a null invocation (an equivalent of a null RPC call) on the server. Unlike in the Sun RPC case, the null method is invoked on the object reference which adds a small overhead.

Table 2 shows the timings for the various actions related to caching causally consistent objects. In this case, two clients executing on the two SPARCstations create objects and access each other's objects. Clearly, invocations on a cached object are very fast - a

[3] An object server provides an address space for the distributed object.

Null Sun RPC	2.35
Null Proxy Invocation	2.7

Table 1: Null RPC and Proxy invocation timings (in milliseconds)

Creating a cacheable object	3.8
Caching an object	8.0
Method invocation on a cached object	0.02
Validating a cached object	6.2

Table 2: Times (in milliseconds) for operations on cached objects

simple invocation on a cached object takes 20 microseconds whereas the cost of a null proxy invocation is 2.7 milliseconds. This might be expected and we now proceed to examine the overheads of caching.

The cost to create a cacheable object[4] is 3.8 milliseconds. This time is dominated by the time for an inter-process communication (IPC) between the client and the corresponding process cacher. This is required for the process cacher to create meta-data on the object copy that the process cacher can use while servicing requests for the object. Validating an object copy takes 6.2 milliseconds. Validating an object copy includes acquiring the object's new state from a valid copy elsewhere in the system. For example, in the experiments conducted, when a client accesses an invalid copy and determines that validation is necessary, it communicates with the consistency cacher on the node on which the object was created. The consistency cacher communicates with the process cacher corresponding to the client process that created the object (and therefore has a copy of the object) and receives the object state. The consistency cacher then returns the object state (and the associated timestamps) to the requesting client. When a valid copy is added to the cache, cached object copies that are found to be mutually inconsistent with the incoming copy are invalidated and the local clock is updated.

[4]A cacheable object is an instance of a class that is a descendant of the Cached class.

Table 2 shows that caching an object copy takes 8 milliseconds. The actions required for caching an object copy are similar to the ones required for validation except that the client also informs the process cacher about it (one additional IPC). Once again, this is because the process cacher needs to maintain meta-data regarding the object that has been cached in order to service later requests for the object. We would like to point out that invalidations are local and take an insignificant amount of time.

6 Discussion

We saw in the previous section that caching considerably reduces latency in accessing shared distributed objects when applications exhibit a reasonable amount of locality of reference. One of the goals of our system is scalability. There are many aspects of scalability. We believe that by allowing applications to use weaker consistency models whenever applicable, the overall scalability of the system is improved. Towards this end, we provide flexible notions of state consistency in our system. Another aspect of scalability that can be of concern is the size of the vector timestamps that we associate with each object copy. Various efficient implementations that reduce the space overhead considerably are outlined in [32, 36]. Below, we comment on our experiences with the present CORBA specification and UNIX.

Passing object reference by value: CORBA compliant systems have the advantage over many commercial RPC systems in that they allow object references to be manipulated as first-class values in a straight-forward manner. In particular, an object reference can be sent as an argument of an invocation on an object server. Thus, CORBA compliant systems provide for network-wide references, and support distributed object reference semantics. However, this does not always provide the required semantics. For example, an application might want to pass "an object reference" by value just like pointer structures are passed by value. For example in Flex, many method invocations in the caching framework take a vector timestamp as one of the arguments. We have defined a VectorTimeStamp IDL interface and we will call the instances of this class as vector timestamp objects. Now, suppose that we want to pass the vector timestamp arguments as references to objects of type VectorTimeStamp in invocations on the object server. Firstly, we need to activate an object server to service the vector timestamp object sent as an argument. Moreover, every invocation on such an object reference (argument) would be an inter-address space invocation which we clearly want to avoid.

Ideally, we would like to have the choice to pass an object reference argument by "value" as in [6, 16, 25, 17]. The object structure should be linearized, sent over the wire, reconstructed at the other end and finally, we should be able to make an object invocation locally at that end.

Transport interface: CORBA specifications include synchronous, asynchronous, and one-way remote object invocations. From our experience, we found that *multicast* is an important mechanism that is useful in consistency maintenance in distributed object systems. For example in Flex, each time an object copy is validated, the node cacher at the owner node can potentially contact node cachers on all nodes on which clients contain copies of the object. This is a perfect candidate for exploiting multicast communication. Though a system level multicast may be available, it cannot be used because the CORBA ORB API does not support it. In [24], the authors argue the need for a multicast interface in the context of object groups.

UNIX and object-oriented technology: We mentioned in section 3 that we use virtual memory (VM) mechanisms for object faulting and access detection. This does lead to some compromise on the object oriented aspect of system development. For example, consider the following scenario. A client faults on a causally consistent object and experiences a segmentation fault. The object fault handler code is defined in the *Causal* class of the consistency framework. The handler returns with the new object state. At this point, the local object copy needs to be created or updated using the received object state. The *readFromString* method in the *Cached* class (or overriding method in the object's class) defines the code that needs to be used to update the object copy's state. However, the handle to the object that is obtained as a result of a segmentation fault is a pointer to an object of type **char**. Since the reference is not appropriately typed, the *readFromString* method cannot be invoked. Our solution, which is outlined below, departs from object technology. Each process maintains a *function table* that contains the address of the *readFromString* method of every type of object for which the client has a reference. When a client creates an object reference, the constructor of the corresponding class invokes the constructor of the *Causal* class with one of the arguments being the address of the *readFromString* method. This address is stored in the function table. Later, when object state is received as a result of an object fault, the stored address of the corresponding *readFrom-*

String method is used to update the local object copy. This could be used as a reason to adopt software based solutions instead of VM based schemes for object faulting and access detection.

Caching as an object service: The Object Management Architecture guide [29] includes a reference model which identifies and characterizes the components, interfaces, and protocols that compose the OMA. The reference model consists of four major parts - the object request broker (ORB), object services, common facilities, and application objects. The ORB enables objects to transparently make and receive requests and submissions in a distributed environment. Object Services is a collection of services (interfaces and objects) that support basic functions for using and implementing objects. Common Facilities is a collection of services that provide general purpose capabilities useful in many applications while application objects are objects specific to particular end-user applications.

As we pointed out earlier, it is neither desirable nor possible to cache all objects. Furthermore, different consistency guarantees may be required of cached copies. Thus, caching should be provided as a Common Object Service and not as part of the ORB. The differences in object caching which manifested in the design of the consistency framework can serve as the interface of the object service itself. We discussed an extension to the IDL while arguing the need to pass objects by value. We also pointed out a desirable enhancement to the interface that CORBA provides to its applications when we argued about the need for multicasting. We also broached upon the issue of providing library object adaptors which are linked with the client applications and which help find implementations of objects that get cached. Clearly more issues remain to be explored, particularly in the realm of identifying interactions with other object services. Our intention in this paper is only to provide a data point in a discussion of issues related to caching in CORBA.

7 Concluding Remarks

In this paper, we described the design and implementation of Flex, a scalable and flexible distributed object caching system, on top of a CORBA compliant system running on the UNIX operating system. An important feature of Flex is that it supports flexible notions of object state consistency thus improving system performance through the use of weaker consistency levels when applicable. We presented issues that arise in an object caching system. We also argued how some of these issues led us to believe that object caching should be provided as a Common Object Service and not as part of the ORB itself. Our implementation experience indicates that object caching can considerably reduce latency in accessing shared distributed objects when applications exhibit a reasonable amount of locality. In particular, our experiments show that the overhead of caching an object can be amortized over less than four method invocations.

Acknowledgements: We would like to thank the referees of this paper for their many valuable comments. We would also like to thank Ajay Mohindra who helped clarify many implementation issues in DSOM (IBM's CORBA compliant system) to one of the authors during the author's summer internship at IBM T. J. Watson Research Center.

References

[1] M. Ahamad, G. Neiger., J. E. Burns,

P. W. Hutto, and P. Kohli. Causal memory: Definitions, Implementations and Programming. *Dist. Computing*, Vol. 9, 1995.

[2] M. Ahamad, F. J. Torres-Rojas, R. Kordale, J. Singh, S. Smith. Detecting Mutual Consistency of Shared Objects, *Proc. of International Workshop on Mobile Systems and Applications*, December 1994.

[3] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive software cache management for distributed shared memory architectures. In *Proc. of the 17th ISCA*, 1990.

[4] B. N. Bershad, T. E. Anderson, E. D. Lazowska, H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. on Comp. Sys.*, May 1991.

[5] B. N. Bershad, M. J. Zekauskas, W. A. Sawdon. The Midway distributed shared memory system. *COMPCON*, Spring 1993.

[6] A. Birrell, G. Nelson, S. Owicki, and E. Wobber. Network Objects. *ACM Symp. on Oper. Sys. Principles, 1993.*

[7] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. *ACM SIGMOD*, 1994.

[8] K. M. Chandy, and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. on Comp. Sys.* Feb 1985.

[9] D. R. Edelson. They're Smart, but They're Not Pointers. *USENIX C++ Conference*, 1992.

[10] I. R. Forman, S. Danforth, and H. Madduri. Composition of before/after metaclass in SOM. In *Object Oriented Programming Systems, Languages, and Applications*, 1994.

[11] C. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Australian Computer Science Conf.*, 1988.

[12] Fresco Sample Implementation Reference Manual. X Consortium Working Group Draft. Version 0.7. April 1994.

[13] A. L. Hosking and J. E. B. Moss. Protection traps and alternatives for memory management of an object oriented language. *ACM Symp. on Oper. Sys. Principles.'93.*

[14] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham. Scale and performance in a distributed files system. *ACM Trans. on Comp. Sys.*, Vol. 6, (Feb).

[15] IBM, 11400 Burnet Road, Austin, TX. *SOMObjects Developer Toolkit Users Guide*, Oct. 1994.

[16] B. Janssen, M. Spreitzer, D. Severson. Inter-Language Unification. Xerox PARC. URL: *http://www.parc.xerox.com/Projects.html*

[17] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Trans. on Computer Systems*, Feb 1988.

[18] P. Keleher, A. L. Cox, S. Dwarkadas, W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Winter USENIX Conf.*, 1994.

[19] P. Kohli, M. Ahamad, and K. Schwan. Indigo: User Level Support for Building

Distributed Shared Abstractions, *Fourth Symp. on High Performance Dist. Computing*, August 1995.

[20] R. Kordale, and M. Ahamad. A scalable technique for implementing multiple consistency levels for distributed objects. *Proc. of the Intl. Conf. on Distributed Computing Systems (ICDCS)*, May 1996.

[21] L. Lamport. Time, clocks and the ordering of events. *Comm. of the ACM*, July 1978.

[22] K. Li, and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Comp. Sys.*, Nov 1989.

[23] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, November 1992.

[24] S. Landis, and S. Maffeis. Building reliable distributed systems with CORBA. In *Proc. of USENIX Conference on Object-Oriented Technologies and Systems(COOTS)*, 1995.

[25] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. Powell and S. R. Radia. An Overview of the Spring System. *Proceedings of COMPCON*, Fall 1994.

[26] A. Mohindra, G. Copeland, and M. Devarakonda. Dynamic insertion of object services. In *Proc. of USENIX Conference on Object-Oriented Technologies and Systems(COOTS)*, 1995.

[27] M. N. Nelson, G. Hamilton, and Y. A. Khalidi. Caching in an Object Oriented System. *Intl. Workshop on Object Orientation in Operating Systems (IWOOOS)*, 1993.

[28] OMG Home page. *http://www.omg.org.*

[29] Object Management Architecture Guide, Revision 1.0, OMG TC Document 90.9.1. *http://www.omg.org.*

[30] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the Sprite network file system. *ACM Symposium on Operating Systems Principles*, 1987.

[31] Flex - A fast scanner generator. Information
available at *http://www.ns.array.ca/ipe-1.2/doc/gnu/flex/flex_toc.html.*

[32] M. Raynal, and M. Singhal. Logical time: A way to capture causality in distributed systems.

[33] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. *Intl. Conf. on Dist. Comp. Sys*, 1986.

[34] B. Steensgaard, and E. Jul. Object and native code thread mobility among heterogeneous computers. *ACM Symp. on Oper. Sys. Principles*, 1995.

[35] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *IEEE Symp. on Parallel and Dist. Information Sys.*, 1994.

[36] F. J. Torres-Rojas. Efficient Time Representation in Distributed Systems. Masters Thesis. Georgia Inst. of Technology.

[37] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software Write Detection for a Distributed Shared Memory. *Oper. Sys. Design and Implementation*, 1994.

Asynchronous Notifications Among Distributed Objects

Yeturu Aahlad[1] Bruce E. Martin Mod Marathe[2] Chung Le

SunSoft, Inc.
2550 Garcia Avenue
Mountain View, California 94043 USA

Abstract

Distributed object systems typically support synchronous requests from one distributed object to another. Often, a more decoupled style of communication among distributed objects is appropriate. We describe an object service called *event channel* that decouples distributed object communication. We describe SunSoft's implementation of the event channel and illustrate its use in a stock trading application.

Keywords: distributed object-oriented systems, communication, events, OMG, CORBA, CORBA Services

1.0 Introduction

Distributed object systems[2][5] are emerging to support applications that access objects across distributed, possibly heterogeneous, system boundaries. Such systems define a distributed object model that is mapped appropriately to native concepts in a wide variety of systems.

Communication in distributed object systems typically consists of synchronous *requests*. A request is made by a requestor to a single target object. A request results in the synchronous execution of an operation by the target object. The requestor waits for a reply from the target. For the request to be successful, the requestor, the target and the network must be available. If a request fails because the target or network is unavailable, the requestor receives an exception.

The principal advantages of basing distributed object communication on synchronous requests are that it is easy paradigm for application programming and that it is an efficient and well-understood basic paradigm to implement in distributed object systems.

For many distributed applications, however, a more *decoupled* style of communication is appropriate. Instead of a requestor identifying the target, the requestor need not be aware of the target. Instead of a single target object, there may be multiple targets. Instead of waiting for a response or failure report, the requestor need not block while the targets process the request.

Consider the distributed stock trading application illustrated in figure 1. As stock prices change, stock objects simply supply the new prices to the distributed system. Interested objects consume the data. What the interested objects actually do with the data is really of no interest to the stock objects. One object may, for example, chart a graph of changes in the stock price. Another object may maintain a portfolio of stocks and make investments based on changes in stock prices. If the network partitions, the stock object does not want to be burdened with delivering the data to the interested objects.

A more decoupled communication paradigm between distributed objects allows applications to easily be extended without modifying existing objects. To extend the functionality of the application, new objects can be added that consume the data supplied by the stock objects. The stock objects need not be modified.

The topic of this paper is decoupling object communication in *CORBA-based distributed object systems* using the basic synchronous request paradigm of distributed object systems. There have been some studies related to decoupled communications in distributed systems.

1. Yeturu Aahlad's current address is Platinum Technology, Inc. Trinzic Lab. 555 Dolphin Drive, Redwood City, California 94065

2. Mod Marathe's current address is StrataCom Inc. 1400 Parkmoor Avenue, San Jose, California 95126

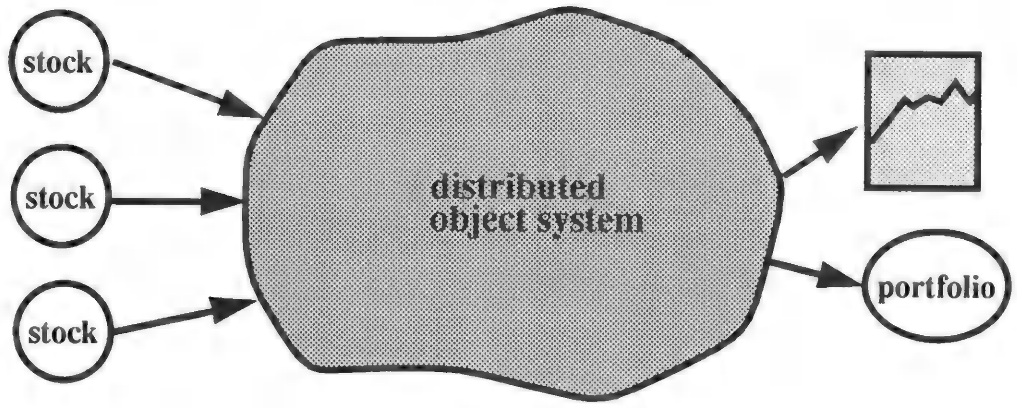


Figure 1. Decoupled object communication in a stock trading application

The Zephyr[1] notification service is a policy rich set of programs providing time-sensitive communications among clients and servers. Zephyr servers provide centralized routing, queuing and dispatching. A reliable and authenticated transmission protocol is supported by the client library. More advanced communication services are provided as additional layers.

In [11], the authors describe a simple and useful system that supports registration and notification of events in an RPC system. Third party services can also be introduced into the chain of notifications allowing for more complex functionality.

Our focus is events in CORBA-based distributed object systems. We first proceed with an overview of the principles of distributed object systems. We then describe an *event channel*, an object that decouples the communication between objects in a distributed object system, motivating our design based on the principles of distributed object systems. Finally, we describe our experience implementing event channels and using them in distributed applications.

2.0 Distributed Object Systems

Distributed object systems embody the following principles:

2.1 All entities are modeled as communicating objects

In a distributed object system all entities are modeled as *objects*. Although the systems being bridged by a distributed object system may include entities that are not objects, they are not available across system boundaries unless they can be presented as objects.

Objects are accessed by clients using object references. Object references can be passed from one object to another. An object holding an object reference for a target object can make a request for its services. The distributed object system provides location transparency to clients. The request may be local or it may cross heterogeneous system and administrative boundaries. It is the distributed object system's job to mask the differences from the client object.

2.2 Interfaces define objects

In a distributed object system, objects are defined by their *interfaces*. An interface specifies a set of operations that defines the behavior of the object. The implementation of an object is separate and invisible; clients cannot depend on implementation properties, such as programming language, transient or persistent representation of the object. There can be multiple implementations of an interface. An interface does not imply any particular implementation(s), and a new implementation may be added at any time. (See [3], [9] for more discussion of separating interfaces from implementations.)

2.3 Federated systems

Distributed object systems are *federated* systems. Existing, disconnected heterogeneous systems are connected and made to interoperate. Gateways mask differences between systems by implementing mappings between concepts in each system, including interfaces, object models, object references and name spaces.

Distributed object systems have the potential of connecting large numbers of objects across system and administrative boundaries. There should be no limit to this; that is, the system should scale.

Federation and scalability lead to truly distributed system objects. Services that depend on a single, centrally administered repository of information are not acceptable. In particular, there is no authority, even a distributed one, that has information about all objects or even part of the information about all objects of one type. Instead, federated services are connected to other instances of the same service to widen their scope of discourse.

2.4 Distributed object systems are open

Heterogeneous systems consist of components that typically come from a variety of suppliers. In order for the components to interoperate, the interfaces between the components need to be standard. Implementations of the components, however, need not be standard. Different suppliers can each provide implementations of a service supporting a standard interface.

The Object Management Group

The Object Management Group (OMG) is promoting standards for distributed object systems among system software vendors. The OMG has currently defined two sets of standards, known as CORBA and COSS. CORBA is the core communication mechanism which all OMG objects use: it enables objects to issue requests of each other. COSS provides standard services that support the integration of distributed objects.

CORBA

The Common Object Request Broker Architecture (CORBA) [6] defines an interface definition language (IDL) for distributed objects. The language allows designers to specify interfaces as a set of operations and attributes. The language supports subtyping of interfaces. A function can be passed an object that supports a subtype of the interface expected by the function.

The CORBA defines object references. Object references are typed by interfaces specified in IDL. Object references unify access to objects. The client using the object cannot tell if the object being accessed is local or remote, who implements the object, or how it is implemented.

The CORBA also defines an interface repository. The interface repository contains descriptions of IDL interfaces and data types. Such descriptions can be accessed at run time to implement type-safe interobject communication. Federating CORBA compliant systems requires correlating the interfaces in different interface repositories.

CORBA Services

The CORBA Services Specifications (COSS) [7] defines a set of services for distributed object systems. The services are specified in OMG IDL and are intended to operate in CORBA environments. Currently, COSS defines a name service for mapping human readable names to object references, a persistence service for persistently representing object state, an object life cycle service for creating, copying, moving and removing objects, a relationship service[4] to support graphs of distributed objects, transaction and concurrency control

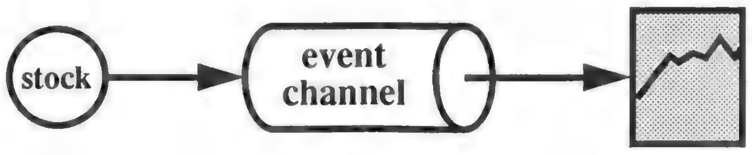


Figure 2. An event channel is an object service that decouples communication among distributed objects.

```
interface PushConsumer {

   void push(in any data);
       raises(Disconnected);

   void disconnect_push_consumer();

};
```

Figure 3. The PushConsumer interface.

services to implement atomic access to objects, an externalization service to externalize and internalize objects and an event service to decouple object communication. The COSS event service is based on the event service described in this paper.

3.0 Event channels

An event channel is an object that decouples object communication. Figure 2 illustrates the stock object and the chart object communicating using an event channel. The stock is the *supplier* of the event and the chart is the *consumer* of the event. An event channel is *both* a consumer and a supplier of events. In figure 2, the event channel consumes the events supplied by the stock object and then supplies those events which are consumed by the chart object.

3.1 Communicating events

Event communication is achieved via standard interobject requests. The event has data associated with it.[1] There are two styles of event communication, the *push* style and the *pull* style. The push style is similar to the *EventCatcher* notification style of [11].

In the push style, consumers support the *PushConsumer* interface, given in figure 3. Suppliers initiate the communication by invoking the *push* operation on the consumer. The *disconnect_push_consumer* operation terminates the event communication.

In figure 2, if the event channel is consuming events in the push style, it supports the *PushConsumer* interface; the stock invokes the *push* operation on the channel. Similarly, if the chart object is consuming events in the push style, it supports the *PushConsumer* interface; the channel invokes the *push* operation on the chart.

1. The data is of the CORBA IDL data type any. See section 3.5 for more discussion of data types.

In the pull style, suppliers support the *PullSupplier* interface given in figure 4. Consumers initiate the communication by invoking a *pull* operation on the supplier. Suppliers support two kinds of pull operations for returning events. The *pull* operation blocks until an event is available. The *try_pull* operation returns immediately, returning an event if one is available. The *disconnect_pull_supplier* operation terminates the event communication.

In figure 2, if the stock object is supplying events in the pull style, it supports the *PullSupplier* interface; the event channel invokes the *pull* operation on it. Similarly, if the event channel is supplying events in the pull style,

```
interface PullSupplier {

   any pull()
       raises(Disconnected);

   any try_pull(out boolean has_event)
       raises(Disconnected);

   void disconnect_pull_supplier();

};
```

Figure 4. The PullSupplier interface.

it supports the *PullSupplier* interface; the chart object invokes the *pull* operation on it.

In figure 2, the stock object can supply events to the event channel in either push or pull styles and the chart object can *independently* consume events from the event channel in either push or pull styles.

The two styles of event communication are very flexible. Push-style event communication is driven by the supplier of events, whereas pull-style event communication is driven by the consumer of events. A push-style consumer of an event channel does not have to actively solicit events. A pull-style consumer, on the other hand, has control over when an event is delivered to it. A push-style supplier to an event channel does not have to buffer events; it simply pushes events to the event channel as they happen. A pull-style supplier, on the other hand, can control the buffering policy.

Multi-way, anonymous communication

An event channel can supply multiple consumers. As illustrated in figure 5, the event channel supplies events to both the chart and the portfolio objects. The event

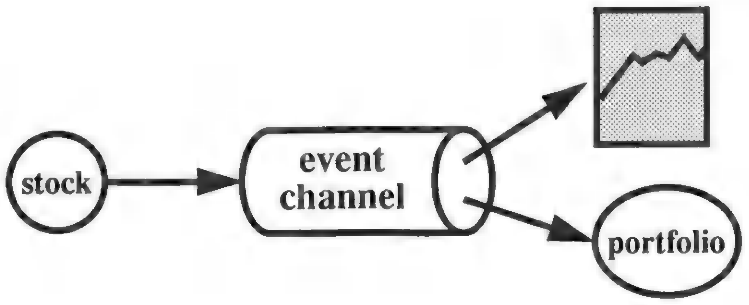


Figure 5. An event channel with multiple consumers.

channel independently communicates with the chart object and with the portfolio object in either push or pull styles.

The event channel makes the communication *anonymous*. The stock object is unaware of the consumers. Structuring a distributed application around anonymous event communication makes it easily extensible. The portfolio functionality was added to the application in figure 5 without modifying the stock object. The stock object continues to supply events, unaware of the consumers.

An event channel can also consume events from multiple suppliers. As illustrated in figure 6, multiple stocks can independently supply events to the same event channel; all consumers receive events from all suppliers.

When there are multiple suppliers, the suppliers of an event are anonymous. If the consumers need to distinguish the multiple suppliers, it can be done in event data.

3.2 Scoping events

So how many event channels are there? Should the cloud labelled "distributed object system" in figure 1 just be replaced with a single event channel? Clearly not; a single event channel for all events in the distributed object system becomes a bottleneck, does not scale and does not federate. Furthermore, all consumers would receive all of the events, most of which are not of interest.

An event channel provides a *scope* for events. There is no need for unrelated application domains to share an event channel. By definition, the consumers of one event channel will not receive the events supplied by another.

In practice, we have found scoping events by supplier to be an effective technique for organizing event communication.

Figure 7 illustrates scoping events by supplier in the stock application. Each stock object supplies events to

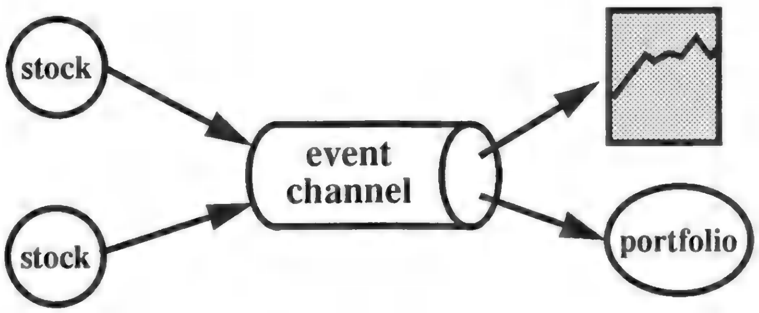


Figure 6. An event channel with multiple suppliers and multiple consumers.

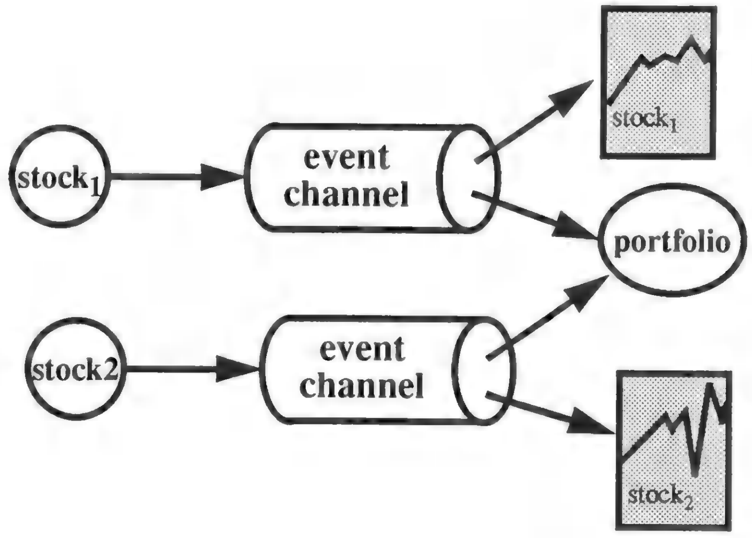


Figure 7. Scoping events by supplier

its own event channel; stock objects do not share event channels. A chart object consumes events from a single event channel since it charts a single stock. The portfolio object, on the other hand, consumes events from multiple event channels, since it makes trading decisions based on multiple stocks.

3.3 Filtering events

Even within the scope of a single event channel, not all consumers want to consume all events. In the stock example, the chart object wants to consume all changes in a stock's price, but suppose the portfolio object wants to consume stock price changes only when the price of a stock reaches a certain price. This can achieved with a special event channel called a *filter*. A filter is a specialized event channel that consumes events from another event channel but only supplies events that satisfy some condition. Events that do not satisfy the condition are simply discarded by the filter.

In Figure 8, only stock prices that are above (or below) a certain price are forwarded to the portfolio object; those that are not are discarded by the filter.

Filters are usually lightweight event channels that just evaluate the event data. A typical implementation of a push-style filter only supports a single consumer and makes no attempt to store and retry supplying the event if it has difficulties communicating with its consumer. The filter itself simply fails. The event channel that supplies events to the filter detects the failure and attempts to supply the event to the filter later.

Filters are best configured to be near the event channels they are filtering. This reduces communication costs when the filter consumes but then discards an event.

3.4 Chaining event channels

For a particular event, there might be numerous consumers residing at a remote location. In this case, chaining of two or more event channels can be used to

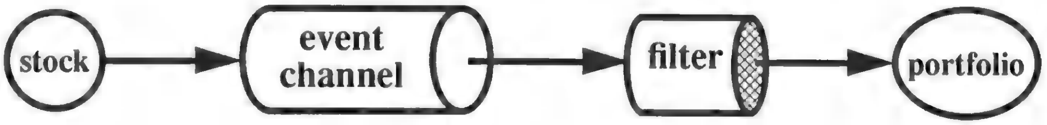


Figure 8. An event channel that filters events for its consumer

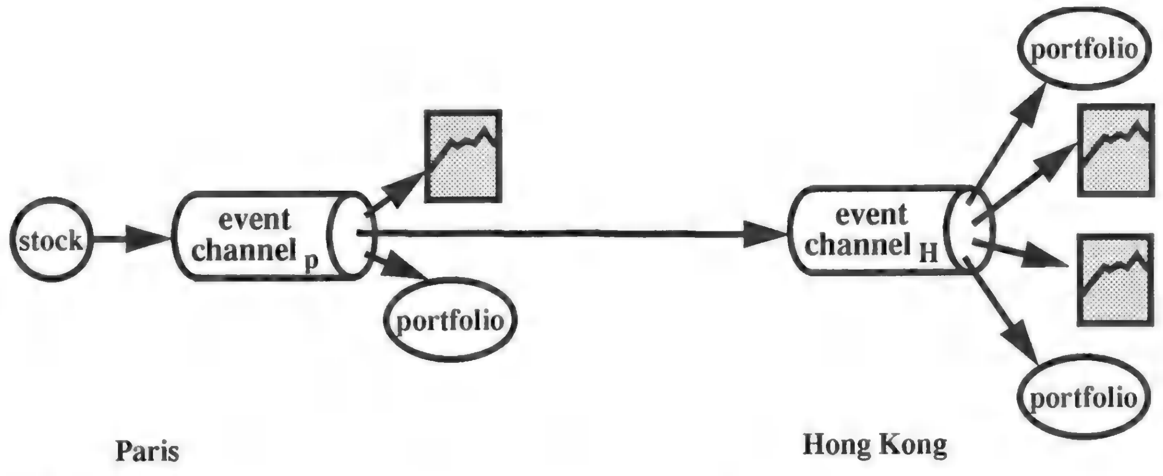


Figure 9. Chaining event channels to minimize communication costs

optimize the delivery of events. For example, in figure 9, there are several consumers of a stock event who are all in Hong Kong. Instead of delivering the same stock event from Paris to each of these consumers across the continents, an event is sent once to the event channel in Hong Kong which then forwards the event to the local consumers.

Event channels support an administrative interface that allows them to be chained. The interface supports operations to obtain consumer and supplier "proxy" objects. Two event channels can be chained in either push or pull styles.

In figure 9 chaining the event channel P in Paris as a push-style supplier and the event channel H in Hong Kong as a push-style consumer is achieved in three steps:

- request a push supplier proxy, p1, from event channel P. The push supplier proxy is a push-style supplier which also supports an operation to connect to a push consumer.
- request a push consumer proxy, p2, from H. p2 is a push-style consumer which also supports an operation to connect to a push supplier.
- connect the proxies p1 and p2 by calling their connect operations.

Event communication proceeds as follows: the stock pushes an event to the event channel P in Paris. P, in turn, pushes the event to its three consumers, one of which is the event channel H in Hong Kong. H, in turn, pushes the event to the two portfolio and two chart objects.

Alternatively, P and H can be chained in pull style by requesting and connecting pull-style proxies.

3.5 Typing events

The data associated with an event is of some programmer defined data type. The *PushConsumer* and *PullSupplier* interfaces given in figure 3 and figure 4 support the passing of the CORBA IDL `any` data type. The `any` data type specifies dynamically typed, self describing data. Any IDL data type can be passed. Event channels usually do not interpret the event data; they just pass it on to consumers. A consumer usually interprets the event data passed to it.

In order to federate multiple distributed object systems, their type spaces must be federated. In CORBA-based systems, this is accomplished by federating interface repositories. An interface repository from one system is merged with the interface repository of another system. Types can be correlated across multiple interface repositories using globally defined repository identifiers.

Rather than define its own type space, the event channel leverages the type system of the distributed object system. It leverages the distributed object type system by using standard interobject requests for communicating with its suppliers and its consumers. As such, event suppliers can communicate with event consumers, via one

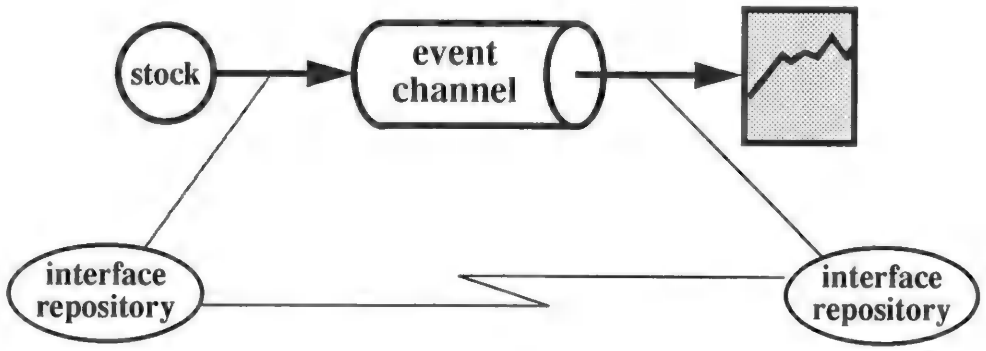


Figure 10. Federating interface repositories enables communication between suppliers and consumers across administrative boundaries.

or more channels, in a strongly typed fashion, even across federated system boundaries.

4.0 Implementing Event Channels

The IDL interfaces of the event channel define basic operations which provide support for asynchronous event communication. These interfaces are generic in the sense that they allow for a wide range of implementations in a wide range of computing environments. For example, one implementation of an event channel might choose to provide reliable delivery of events while another might provide less reliable, but faster delivery of events. One implementation might be optimized for execution in a local program, while another implementation might be designed for operation in a distributed environment of shared persistent objects.

A particular event channel implementation defines certain policies:

acquisition policy

This policy defines when an event channel obtains an event from its suppliers.

delivery policy

This policy defines which events are delivered to which consumers and when. This policy is especially of interest in a distributed environment in which consumers may be unavailable due to partial failures of the system.

discard policy

This policy defines when an event is discarded by an event channel.

persistence policy

This policy defines what part of an event channel's state is persistent and when an event channel saves changes to its persistent state.

4.1 Our implementation

At SunSoft we have implemented the event channel as part of SunSoft's NEO product[10]. NEO includes a CORBA-based, shared, persistent, multi-threaded[8] distributed object environment with location transparency and automatic activation and deactivation of objects. Our event channel implements the acquisition, delivery, discard and persistence policies in that environment as follows:

acquisition policy

An event channel accepts all events pushed at it. This eliminates the need for push-style suppliers to buffer their events by letting them leverage the event channel's buffer. Even if several events happen in the time it takes to push one event to the event channel, a multi-threaded supplier can concurrently push out all events as they occur.

An event channel pulls events from a pull-style supplier only in response to demand from consumers. The event channel dedicates a thread to each pull-style supplier. The thread invokes the *pull* operation on the pull-style supplier. The channel stays at most one event ahead of demand with respect to each pull-style supplier.

Since the event channel operates in a distributed environment, a pull-style supplier of an event may not be available. In such a case, the event channel retries with exponential back-off. If the event channel determines that the pull-style supplier no longer exists or has permanently failed, it disconnects the supplier from the channel.

Slow pull-style suppliers do not affect the service to other suppliers because the event channel isolates suppliers from each other by serving each supplier with a separate thread. A thread blocked in a *pull* request on a supplier does not impede event acquisition from other suppliers.

delivery policy

A consumer subscribes to events by connecting to and disconnecting from an event channel. When an event channel receives an event, it supplies it to all consumers that are currently connected to it. For pull-style consumers, the event is available via a *pull* operation. For push-style consumers, the event channel invokes the *push* operation on the consumer and passes the event data as an argument.

Since the event channel operates in a distributed environment, a push-style consumer may not be available. In such a case, the event channel retries with exponential back-off. If the event channel determines that the push-style consumer no longer exists or has permanently failed, it disconnects the consumer from the channel.

Slow push-style consumers do not affect the service to other consumers because the event channel isolates consumers from each other by serving each consumer with a separate thread. A thread blocked in a *push* request on a consumer does not impede event delivery to other consumers.

All consumers of an event channel receive events in the same order. Furthermore, the consumers receive the events from a single supplier in the order in which they were supplied to the channel.

Events are delivered to consumers at most once. An event may not be delivered to a consumer due to the channel's persistence and discard policies.

discard policy

An event channel has a finite buffer whose size is determined when it is created. When the buffer overflows, the oldest event in the buffer is discarded to make room for the newest.

Another seemingly reasonable approach is to discard the newest event when there is an overflow. However, this policy suffers a serious problem. Consider an event channel with several consumers, one of whom is very slow. When the buffer fills with events not yet delivered to this consumer, subsequent events will be discarded, causing a degradation of service to all other consumers. With our chosen policy, only the slow consumer will lose events. It is a good design principle to ensure where possible that service to well-behaved clients will not degrade because of ill-behaved clients.

persistence policy

Most objects in CORBA-based environments are persistent; the object request broker transparently activates objects upon client demand and deactivates objects to free up computational resources. As such, an event channel persistently remembers the object references of its suppliers and of its consumers. Thus a supplier, a consumer, or the event channel itself, can deactivate and the connection between suppliers and consumers remains valid.

In contrast, it is possible to justify a wider range of policy towards event data. Communication speed and reliability requirements of applications vary widely. Therefore, in our implementation, the creator of an event channel can choose at creation time whether the events are transient or persistent.

By definition, transient events are not saved to the event channel's persistent state. As such, transient events will be lost when an event channel deactivates. If persistent events are chosen, the creator may choose from among a range of save policies, determining the appropriate trade-off between communication speed and reliability. The most conservative policy saves the event data to the event channel's persistent state whenever a new event is supplied to the channel. Alternatively, the

TABLE 1. Push events throughput

Metric	Transient events events/sec 128 byte	4 byte	Persistent events events/sec 128 byte	4 byte
same machine push throughput. Supplier, consumer and event channel are separate processes on the same machine.	66	71	58	65
same LAN push throughput. Supplier, consumer and event channel are on separate machines on the same LAN.	100	144	90	125
same LAN multicast push throughput. Supplier, 5 consumers and event channel are on separate machines on the same LAN.	29	51	21	46

time interval between saves and/or the number of events delivered between saves can be specified by the creator of the event channel.

4.2 Performance of our implementation

The two primary performance metrics for the event channel are throughput and latency. The factors that impact these metrics are:

1. persistence of events
2. the size of data carried in the event
3. the number of consumers and suppliers
4. the location of the supplier(s), the event channel and the consumer(s)
5. push versus pull style

For the sake of brevity, we do not consider the effects of push versus pull style, nor multiple suppliers.

4.2.1 Measurement Methodology

We measure throughput by pushing events as fast as we can into an event channel which has a buffer of 100 events, without the event channel losing events. We then report the number of events delivered to the event channel per unit time in the steady state as the throughput. The latency is measured by recording the machine's high resolution time, pushing an event[1], recording the high resolution time when the event is received by the consumer and then repeating this entire measurement after a short delay. The difference in the two times is the latency. This procedure has the obvious problem that the clocks on different machines are not synchronized enough to allow accurate measurements at the millisecond level. We employ two methods to work around this problem. First, we have a test in which the supplier and consumer processes are located on one machine and the event channel is on another machine. Since the times are recorded in the supplier and consumer, and since both of these are running on the same machine and thus using the same low-level timer, the latency measurement is accurate. When the supplier and consumer are on separate machines, we use an application level echo protocol to calibrate the time skew between the supplier and each consumer to adjust the measured latency value.

All the measurements were conducted using Sparc 10/51 single processor workstations running NEO 1.0 and Solaris 2.4. When multiple workstations were used, they were interconnected with a lightly loaded 10 Mbit/sec Ethernet. The measurements used a sample size of 500 events.

4.2.2 Event Channel Throughput

Table 1 shows the throughput for push events. The multicast measurements use 5 push consumers on separate machines. Within each category of transient or persis-

1. The time required to place the event data into an "Any" datatype is not included in the measured latency. However, the time for marshalling and unmarshalling this data is included.

TABLE 2. Push events latency

Metric	Transient events millisec 128 byte	Transient events millisec 4 byte	Persistent events millisec 128 byte	Persistent events millisec 4 byte	Interobject requests millisec 128 byte	Interobject requests millisec 4 byte
same machine push latency. Supplier, consumer and event channel are separate processes on the same machine.	11	9	13	9	6.2	6.1
same LAN push-push latency. Supplier and consumer are on separate machines and the event channel is on a separate machine on the same LAN.	11.4	10	12.1	10	6.0	5.7
same LAN multicast push-push latency. Supplier and 5 consumers are on separate machines and the event channel is on a separate machine on the same LAN.	17	16	19.4	16	31	28

tent events, we show measurements for events containing 128 bytes of data and 4 bytes of data.

One interesting observation is that the throughput is higher when using a LAN than when all three participants (supplier, event channel and the consumer) are on the same machine. This is not hard to explain - on a single machine, the three participants contend for the single processor and memory whereas when using a LAN, there are 3 different machines providing three times the computing resources.

Another interesting comparison is between having a single consumer versus having 5 consumers. In a simple-minded event channel implementation, multicast throughput would be expected to be one-fifth for 5 consumers. However, since our event channel is implemented using a separate thread for each consumer, we get the advantage of concurrent operation even on a single processor running the event channel.

Since the persistent events require more processing in the event channel and also require periodic[1] saves of the persistent database, their throughput is lower than that of transient events. Also, the size of the event data also affects the throughput substantially.

Transient events are more efficient than persistent events. However, because of the design of the persistent event channel, you can trade off performance against the reliability provided by persistent events. When an event channel is created, the two parameters "commit_time" and "commit_count" can be set based on the application needs. The larger these values, the faster a persistent event channel performs. However, event channels with a high commit_time or commit_count can lose more events in the case of the event server crash.

4.2.3 Event Channel Latency

Table 2 shows the latency measurements for the same factors as in table 1. The latency does not vary much between the same-machine and same-LAN cases. This is because for the latency measurements, we issue events one at a time so the three participants are not active simultaneously. Again, persistent events require slightly more time as do events containing more data bytes. The benefits of a multi-threaded event channel are obvious since the latency for delivering an event to 5

1. Persistent events were configured to save event data every 100 events or 60 seconds whichever is earlier.

consumers is only slightly larger than the latency for a single consumer.

It is interesting to compare the latency through the event channel with the time required to make direct interobject requests. (The latter time is shown in the columns labeled "Interobject requests in table 2.) The event channel implementation takes about twice the time needed to make a direct interobject request which is quite reasonable since it does take two interobject requests (supplier to event channel and event channel to consumer) to deliver the event. When sending an event to 5 consumers, the event channel has a lower latency compared to the supplier making 5 interobject requests one after another to the 5 consumers. This clearly shows the performance gains obtained due to the multithreaded implementation of the event channel.

4.3 Experience using event channels

To date, several distributed applications have been built at SunSoft that use the event channel we have described here, including the stock trading application we have used to exemplify the event channel. Typically, we run this application with a few hundred stocks, a few hundred portfolios and dozens of concurrent presentations. Because of the scoping and scalability properties of the event service and the distributed object system, the application has operated on tens of thousands of stock objects simply by providing more computational and storage resources to the objects.

5.0 Conclusions

We have described how interobject communication in a distributed object system is decoupled using an *event channel*. Communication with an event channel is achieved via standard interobject requests.

The event channel implements the decoupling. In particular, it implements:

- asynchronous communication

  An event channel does not block a supplier while delivering an event to a consumer. Instead, it buffers the data, returns control to the supplier and asynchronously delivers the event to the consumers.

- failure handling

  Because event communication is asynchronous, the event channel handles communication failures by retrying to deliver (or acquire) the event at a later point in time. This relieves the supplier from having to include complicated error handling logic.

- many-to-many communication

  Many suppliers of events can share the same event channel. Similarly, many consumers of events can share the same event channel.

- anonymous communication

  A supplier does not direct request to a particular consumer, but rather it supplies events to an event channel. Anonymous communication makes a distributed application more extensible. Additional consumer objects that extend the functionality of the application can be added without disturbing the supplier objects.

We have also described how event channels provide a scope for events in a distributed objects system, how event channels can be chained to optimize communication costs, how events can be filtered and how strongly typed events are supported.

We have described SunSoft's implementation of the event channel, its performance and our use of it in a simple stock trading application.

6.0 References

[1] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohn and William E. Sommerfeld, "The Zepher Notification Service," In *Proceedings of the Winter USENIX conference*, Winter, 1988.

[2] Bruce E. Martin, Claus H. Pedersen and James Bedford-Roberts, "An Object-Based Taxonomy of Distributed Computing Systems." In *Readings in Distributed Computing Systems*, published by IEEE Computer Society Press. edited by Mukesh Singhal and Thomas L. Casavant. Also in *IEEE Computer* special issue on distributed computing systems, edited by Mukesh Singhal and Thomas L. Casavant, August, 1991.

[3] Bruce E. Martin, "The Separation of Interface and Implementation in C++." In *The Evolution of C++*, edited by Jim Waldo, published by MIT press. Also in *Proceedings of the 3rd USENIX C++ Conference*, April, 1991, Washington, D.C.

[4] Bruce E. Martin and R.G.G. Cattell, "Relating Distributed Objects." In The Proceedings of the 20th VLDB Conference, September, 1994, Santiago, Chile.

[5] John R. Nicol, C. Thomas Wilkes and Frank A. Manola, "Object Orientation in Heterogeneous Distributed Computing Systems," *IEEE Computer*, June, 1993.

[6] Object Management Group, "The Common Object Request Broker: Architecture and Specification, version 2", March, 1995

[7] Object Management Group, "CORBAservices: Common Object Services Specification", OMG Document Number 95.3.31, March 31, 1995.

[8] M. L. Powell, S. R. Kleinman, S. Barton, D. Shah, D. Stein and M. Weeks, "Sun{OS} Multi-thread Architecture", In *Proceedings of Usenix Winter conference*, Winter, 1991.

[9] Alan Snyder. "Encapsulation and Inheritance in Object-oriented Programming Languages", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications* (OOPSLA). Association of Computing Machinery, 1986.

[10] SunSoft, Solaris NEO Operating Environment, Product Overview, 1995.

[11] Jim Waldo, Ann Wollrath, Geoff Wyant and Samuel C. Kendall. "Events in an RPC Based Distribtured System." In *Proceedings of the 1995 USENIX Annual Technical Conference*.

Preliminary Design of ADL/C++ — A Specification language for C++

Sreenivasa Rao Viswanadha[*]
Department of Computer Science
State University of New York
Albany, NY 12222, USA
sreeni@cs.albany.edu

Sriram Sankar
Sun Microsystems Laboratories
2550 Garcia Avenue
Mountain View, CA 94043, USA
sankar@eng.sun.com

Abstract

ADL/C++ is a specification language for associating behavior specifications as post-conditions with C++ (function) declarations. The language includes a clean subset of the C++ expression language and the semantics is relatively simple and informal. Most of the important features of C++ like member functions, virtual members, constructors, inheritance and exceptions can be specified in ADL/C++. In addition to being semi-formal documentation, ADL/C++ specifications can also be used for testing C++ implementations. We developed a method for validating tests of C++ implementations using ADL/C++ specifications.

1 Introduction

Traditionally specification languages have been designed to give a specification of a computation independent of the implementation language(s) in which the computation is realized. While this approach may have the advantage of giving an independent characterization of algorithms and data structures, it imposes the burden on the users to learn two different languages using possibly two different formalisms and two different paradigms (since a specification emphasizes *what* and an implementation emphasizes *how*). For example, the specification language Z [9] uses set-theoretic semantics for specifications and Larch [3] and Tecton [4] are languages based on algebraic specification. But, most imperative programming languages have state-based operational semantics. Perhaps this is the reason why there does not seem to be much use of these specification methodologies in the industry.

[*]This work is supported by funding from Sun Microsystems Laboratories.

A somewhat different approach has been taken in high-level functional and logic programming languages where a single language framework is used for specification and implementation. Our approach has similar goals but is still quite different from these. A subset of the programming language itself (with some extensions) is used to write specifications. This has the advantage that developers do not have to learn different languages, possibly supporting different paradigms. While writing specifications the programmer is encouraged to specify *what* the program does without worrying about implementation details, and for this, a small "clean, *side-effect-free*" subset of the language can be used.

We believe that such an approach will encourage developers to write specifications before working on implementations. Such specifications can be easily used for testing purposes also, since an implementation can be executed with an input data and the output generated along with the test input can be plugged into the specification to see whether the specification is satisfied for that particular test data.

ADL (Assertion Definition Language) adopts this philosophy. It is a family of languages designed to associate semantic information with declarations of variety of languages (C, C++ *etc.*). One of the basic goals of the language is to remain as close to the implementation language as possible, at the same time allowing high-level specification independently of implementations. Another feature of ADL is that specifications written in ADL are checkable, *i.e.*, implementations can be tested against ADL specifications.

There are some inherent limitations with this approach, particularly, that the specification will inherit all the pitfalls and ambiguities of the programming language depending upon the restricted subset of the language being used for specification purposes. But, we have taken care to select a subset of the underlying programming language which is well-understood with minimal ambiguities.

The rest of the paper is organized as follows. In Section 2, we present some background on ADL. Section 3 presents an overview of of ADL/C++ using an example. In Section 4, a somewhat detailed description of the language syntax and informal semantics is given. Section 5 describes how reuse of specifications is supported in ADL/C++. In Section 6, a method for validation of C++ implementations using ADL/C++ specifications is presented. Section 7 gives a comparison with other related work. We conclude in Section 8 with a brief discussion about future work.

2 Assertion Definition Language

ADL is a specification language framework used to describe input-output behavior of software systems. ADL originated as part of a conformance testing system (ADLT) [11] for ANSI C API's. In this system, the runtime behaviors of implementations of API's are compared to their specification written in the ADL framework. These specifications are written in a C like syntax as extensions to C header files. This particular syntax is termed ADL/C (or ADL for C). The ADLT project commenced in early 1993 and has culminated in a robust industry quality environment for conformance testing.

ADLT has a collection of tools developed for ADL/C and a language called TDD, for describing test data. The tools include ACF, the assertion checking function generator that translates post-conditions given in ADL/C into C functions, a "natural" language document generator, and a test coverage analysis tool. Some of these tools are being successfully used by X/Open and NIST as a part of their conformance testing efforts.

We are developing a similar specification capability for C++, namely ADL/C++. ADL/C++ builds on the experience gained from the ADLT project. It is a natural extension of ADL/C for C++, but it also represents a significant improvement in expressiveness and ease of use. A subset of ADL/C++ itself may be used to specify C programs, and we anticipate that this subset will eventually replace the old ADL/C[1] . Other projects currently being undertaken are ADL/IDL (OMG IDL [2]) and ADL/Java™. While this paper focuses on ADL/C++, the general concepts presented here apply equally well to the other language specializations of ADL.

[1]Upward compatibility will be maintained so that the existing ADL/C specifications can work with the new set of tools.

Java™ is a trademark of Sun Microsystems Inc.

3 Overview of ADL/C++

ADL/C++ supports behavior specifications for methods (member functions) and constructors in the form of post-conditions. It also supports specification of inheritance, virtual functions, and exceptions. For this, it uses a *side-effect-free* subset of C++ expression language as described in [10] and extends it with the usual logical operators, bounded quantification, call-state operator (for evaluating expressions prior to a function call), return expressions (to denote return values of functions), and some structuring constructs.

A function behavior is typically partitioned into normal and abnormal behavior by giving the corresponding predicates in terms of the values of the parameters, return values, and exceptions thrown. Then, predicates that describe the state transformations corresponding to each of these behaviors are given. Additional predicates that describe the state transformations and invariants independent of normal or abnormal behavior may also be given.

Consider an example ADL/C++ specification of a function that computes the integer square root of a given number.

```
specification {
   int sqrt(int x)
   semantics
   {
      @x >= return * return;
      @x < (return + 1) * (return + 1);
   };
};
```

Intuitively, the above specification says that *sqrt* is a function that takes an integer and returns a (positive) integer which is the floor of the square root of the input parameter value. Albeit using C++-like expressions, the above specification just says *what* the function *sqrt* should do, but not *how* it should be done[2].

The operational meaning of the above specification is that if *j* is the return value of a call *sqrt(k)* for some (positive) integers j and k, then both the formulas listed in the semantics clause should evaluate to *true* by setting the value of the variable x to k and that of **return** to j. More precisely, the following two predicates should evaluate to *true* :

$$k >= j * j$$
$$k < (j+1) * (j+1)$$

The "@" operator says that the expression following it should be evaluated before a call to *sqrt*. The keyword **return** denotes the return value of a call to *sqrt*.

The complete syntax and semantics of the language are presented in Section 4.

We illustrate the specification of various important features of C++ using the more realistic example presented in Figures 1 and 2 of a specification of a bank account class. The account class has methods to deposit and withdraw amounts. The declaration of the *Account* class can be first developed in a file *account.hh* and then, it can be included in the specification file to give behavior specifications for the various (member) functions.

The *Account::Deposit* method semantics says that it throws an exception if called with a negative value for *amount*. This is specified using the "<:>" operator. Informally, this means that a negative value of the *amount* would result in abnormal termination and in case abnormal termination, the presence of a *NegativeAmount* exception implies that the value of *amount* was negative. In the normal case, the value of *balance* increases by the amount that is deposited.

The *Account::Withdraw* method semantics says that the operation terminates abnormally if either of the exceptions corresponding to insufficient funds in the account or a request to withdraw a negative amount is thrown. Otherwise it terminates normally and the value of *balance* decreases by the amount withdrawn. Then it also says what the values of a thrown exception should be.

The constructor specifies what the initial values of various data members should be in case of normal termination. It also specifies the exceptions thrown in case of illegal initial values for some of the data members.

[2] This is a partial specification because no return value can satisfy the behavior specification if the input value of the parameter x is zero or a negative number. This can be easily fixed using exceptions.

3.1 Functions

Behavior specifications for a function can be written using all the (program) variables that can be used in a definition of that function with the same scope and visibility rules. A specification typically relates the before and after values of the part of the state that it can modify in a particular call. This includes all the global variables, data members (for member functions) and any reference parameters. In addition, it can also specify what exceptions, if any, it can throw and the properties of the values of the exceptions thrown.

A behavior specification for a function can be given inline at the time of declaration, or separately, just like a definition.

Constructors

In semantics for constructors, typically one can specify the initial values for various data members which effectively gives the initial state of the object. In the example, the semantics for the constructor for the *Account* class specifies that the values of the account number, type and initial balance should be equal to the corresponding parameters.

In a specification for a constructor, the values of the data members of the object (being constructed) cannot be accessed in the call-state ("@" operator) because the object would not have been constructed before a call to a constructor.

Virtual Member Functions

If a superclass declares a member function to be virtual, then a call to it using a pointer might actually invoke its implementation in a subclass. Therefore it is logical that semantic description given in a subclass implies the semantics specified in the superclass for a virtual function.

To achieve this, ADL/C++ imposes the semantic restriction that any virtual member function redefined in a derived class should obey the semantics given for it in the base class as interpreted in the context of the base class. This means that the assertions in the derived class are strengthened by those given in the superclass. This, in some sense extends the (mostly syntactic) inheritance mechanism of C++ to inheritance of properties of methods rather than respecifying the properties given in the superclass.

For example, consider the ADL/C++ specification given in Figure 3, of a new bank account class

```
#ifndef ACCOUNT_HH
#define ACCOUNT_HH 1

class Account
{
   public :
      enum { MAX_ACCOUNT_NUM = 100 };
      enum AccountTypes { CHECKING = 1, SAVINGS };

   private :
      const AccountTypes accountType;
      const int accountNum;
      long balance;

   public:
      // Exception classes.
      struct InvalidAccountNum {
         const int num;
         InvalidAccountNum(int i) : num(i) { };
      };

      struct InvalidAccountType {
         const int type;
         InvalidAccountType(int i) : type(i) { };
      };

      struct InsufficientFunds {
         const long bal;
         InsufficientFunds(long i) : bal(i) { };
      };

      struct NegativeAmount {
         const long amt;
         NegativeAmount(long i) : amt(i) { };
      };

      // Interface
      virtual void Deposit(long amount)
         throw(NegativeAmount);
      virtual void Withdraw(long amount)
         throw(NegativeAmount , InsufficientFunds );
      Account(AccountTypes type, int num, long bal = 0)
         throw(InvalidAccountNum, InvalidAccountType);
};

#endif /* ACCOUNT_HH */
```

Figure 1: A Bank Account Class Declaration

```
specification AccountSpec
{
#include "account.hh"

   void Account::Deposit(long amount) throw(Account::NegativeAmount na)
   semantics
   [ abnormal := thrown(na); ]
   {
      (amount < 0) <:> thrown(na);
      normal ==> balance == @balance + amount;
      abnormal ==> unchanged(balance);
   };

   void Account::Withdraw(long amount)
      throw(Account::NegativeAmount na, Account::InsufficientFunds isf)
   semantics
   [ abnormal := thrown(na) || thrown(isf); ]
   {
      (amount > @balance)  <:> thrown(isf);
      amount < 0           <:> thrown(na);

      thrown(isf) ==> (isf.bal == balance);
      thrown(na)  ==> (na.amt == amount);

      if (abnormal) { unchanged(balance); }
      else { balance == @balance - amount; };
   };

   Account::Account(AccountTypes type, int num, long bal)
      throw(Account::InvalidAccountNum ian,
            Account::InvalidAccountType iat)
   semantics
   {
      (num < 1 || num) > MAX_ACCOUNT_NUM    <:> thrown(ian);
      (type != CHECKING && type != SAVINGS) <:> thrown(iat);

      thrown(ian) ==> (ian.num == num);
      thrown(iat) ==> (iat.type == type);

      if (normal) {
         accountNum == num; accountType == type; balance == bal;
      };
   };
};
```

Figure 2: A Specification of a Bank Account Class

```
with [AccountSpec] // Inherit the previous specification
specification NewAccountSpec
{
   class OverdraftAccount : public Account {
      long maxOverDraft;  // Max overdraft allowed (can change)
      long overDraft;     // How much overdrawn

     public :
      OverdraftAccount(AccountTypes type, int num, long bal)
        : Account(type, num, bal + maxOverDraft) {
         overDraft = 0;
      };

      void Withdraw(long amount) throw(Account::NegativeAmount na,
                                       Account::InsufficientFunds isf)
      semantics {
         if (normal && bal < maxOverDraft) {
            overDraft == maxOverDraft - bal;
         };
      };
   };
};
```

Figure 3: A Specification Using Inheritance and Virtual Functions

which inherits from the *Account* class. It allows an overdraft upto a certain amount which can change. Therefore the new *Withdraw* method will say how the value of the variable *overdraft* changes if the account is being overdrawn. Since it is virtual, by our semantics it automatically inherits the specification given in the base class and therefore it is not necessary to specify that part. In this case, the variable *balance* merely denotes the amount available for withdrawal, not the real balance.

Another common use of virtual functions in C++ is the specification of abstract classes. The abstract properties can be specified for the virtual functions in the abstract class and in the implementation classes, these functions would have to obey all those properties automatically because of the semantic constraints imposed by ADL/C++.

This semantics can also be useful to specify the behaviors for specializations if the superclass specification is not overly constrained. For example, the sorting functions specified in Section 5 can be easily specified using C++ inheritance by appropriately making the first sort function a virtual member of a class and then redefining it as a stable sort function in a subclass. This is possible because the original sort function does not say anything about the relative order of elements with equal key and hence the subclass has the freedom to add the extra constraint.

The idea here is to impose some discipline on inheritance (which C++ doesn't) so that some kind of subtyping is achieved. Our current semantics for virtual functions is inspired by the behavioral notion of subtyping described in [6] and a similar scheme called *weak behavioral subtyping* is supported in Larch/C++ [5].

3.2 Exceptions

ADL/C++ allows the specification of exceptions that can be thrown by a function. In the semantic specifications of functions, exceptions can be explicitly used (see the next Section). C++ does not allow exceptions to be named in the throw clause. However, in ADL/C++, they can be named just like other parameters, and can be used in the post-condition. We feel this will make specifications more readable than the corresponding C++ code because now users can name exceptions in a way as to reflect the nature of the exception. In the bank account example above, instead of using separate exceptions, if the class designer decided to throw an exception of type *long* in case *amount* is negative, it can be named appropriately (say,*negativeAmount*) to reflect this fact in the behavior specification. We also extended the C++ exception declaration to specify things like no exception can be thrown and any exception can be thrown.

The semantics of exception specification is that any implementation can throw *at most* those exceptions listed in the throw clause. It can not throw anything that is not mentioned. Note that in C++, the throw list that follows a function declaration is just informational and the function can throw other types of exceptions not listed there.

In the post-condition, tests can be made for the presence of exceptions using the **thrown** operator. Exception values can be used just like normal data values except that it is illegal to use an exception when it is not thrown. Similarly, it is illegal to use an exception in the call state. It is also illegal to use the return value in case an exception is thrown.

In the example, the semantics for *Account::Withdraw* specifies that it can throw *InsufficientFunds* and *NegativeAmount* exceptions to signal abnormal termination. It also says that if the *InsufficientFunds* exception is thrown, then the value of its data member *bal* should be equal to *balance*.

Note that exception conditions need not always be part of the abnormal section of function semantics. Since exceptions can also be used for communication, they can be used as part of normal behavior also. For example, a read method of a file class may throw an exception to signal an end-of-file. But, usually it is not abnormal behavior. On the other hand, if the file is not open, then the exception thrown usually signals abnormal termination.

3.3 Access Control

We envisage the use of ADL/C++ for giving detailed specifications for implementations also, therefore, ADL/C++ ignores all access restrictions. This, we believe, will give the implementor flexibility to specify and test the implementations without having to worry about access permissions. This gives the ability to look at (but not to modify) private data not accessible to a function to do test validation.

4 Syntax and Semantics

Much of ADL/C++ syntax includes C++ syntax, but it also has some syntactic constructs of its own. We will describe the ADL/C++-specific syntax and semantics in detail in the following subsections.

4.1 Specification Structure

adl_specification ::=
    { *with_clause* }
    specification *spec_name*
    "{" (*annontated_declaration*) + "};"

Every specification has a name and consists of one or more annotated declarations, that is, C++ declarations optionally annontated by behavior specifications. At present, we require that the name of the file be same as the name of the specification with **.adl++** as the file name extension. ADL/C++ is case-sensitive just like C++ is.

An ADL/C++ specification can also import other ADL/C++ specifications using the **with** clause.

with_clause ::=
    with "[" (*spec_name*)
        ("," *spec_name*) * "]"

A specification can use all the declarations and any annotations from any of the specifications listed in the **with** clause.

4.2 Declarations

All C++ declarations like typedefs, classes, enums, variable declarations *etc.* can be present in an ADL/C++ specification. To facilitate users to include "real" C++ header files, we allow inline function definitions to be present in ADL/C++ specifications. At present, only function declarations can be annotated to give behavior descriptions.

annontated_declaration ::=
    C++_declaration
  | *auxiliary_declaration*
  | *C++_function_declaration annotation*

C++_declaration is any valid C++ declaration and *C++_function_declaration* is a C++ function declaration without the **throw** clause.

Annotations

An annotation consists of ADL/C++ exception specifications, binding definitions for normal and abnormal behavior of the function, and a list of formulas that describe the behavior of the function.

```
annontation ::=
        { adl_exception_spec }
        semantics
        { behavior_definitions }
        group_expression ";"

adl_exception_spec ::=
        throw { ( exception_list ) }

exception_list ::=
        param_declaration
              ( "," param_declaration ) *
    |   "..."

behavior_definitions ::=
        "[" ( ( normal | abnormal )
              ":=" expression ";" ) + "]"
```

As mentioned earlier, ADL/C++ exception specifications can be named. So, the syntax for this is same as that of a C++ formal parameter declaration. If no exception is specified following the throw clause, then the function can not throw any exception. If there is a "..." in the throw list, then the function can throw any exception.

Note that all the variables used in the annotation part are program variables (and not logical variables).

An annotation for a function is like a post-condition, *i.e.*, all the expressions except call-state expressions (see below) that constitute the group expression will be evaluated using the values of the variables after a call to the function.

The behavior definitions classify the behaviors of the function into normal and abnormal behaviors. If neither normal nor abnormal behavior is defined, then abnormal defaults to thrown(...) and normal defaults to !thrown(...). If only of normal or abnormal behavior is defined, the undefined one defaults to the negation of the defined one. There can be at most one definition each for normal and abnormal behaviors in an annotation.

Auxiliary Declarations

Sometimes, if the interface or class declaration does not contain enough information about the state, then it becomes necessary to have some auxiliary declarations for specification purposes. Since these are needed only for specification, but not a part of the original declarations, ADL/C++ provides a mechanism to declare them as auxiliaries.

```
auxiliary_declaration ::=
        auxiliary "{"
              ( C++_declaration ) * "};"
```

4.3 Expressions

The expression language includes the side-effect-free subset of the C++ expression language. All forms of C++ assignment operators, increment and decrement operators and any overloaded versions of these are excluded. However, ADL/C++ does have the function-call operator. We provided this to facilitate testing. One should be very careful in using function calls as there can be side-effects in the invoked functions. ADL/C++ also has some special operators of its own.

```
expression ::=
        C++_expression
    |   adl_expression

adl_expression ::=
        group_expression
    |   call_state_expression
    |   adl_logical_expression
    |   quantified_expression
    |   thrown_expression
    |   unchanged_expression
    |   behavior_expression
    |   return
    |   normal
    |   abnormal
```

We will now describe the various ADL-specific expressions.

Group Expressions

```
group_expression ::=
        "{" ( binding ) *
            ( labels { tags }
            expression ";" ) * "}"

labels ::=
        ( IDENTIFIER ":" ) *

tags ::=
        "[" IDENTIFIER
            ( "," IDENTIFIER ) * "]"
```

A group expression is a semicolon separated list of expressions. Each of these expressions should be a boolean expression and the type of the group expression is also boolean. The semantics of a group expression is the conjunction of the list of expressions that constitutes the group. Note that the expressions can be evaluated in any order and a ";" does not denote sequencing, but conjunction.

Expressions in a group can be given names using labels. The tags are used by tools for generating reports in case of testing.

Bindings

Since assignment expressions are not allowed in ADL/C++, a set of variables can be declared and initialized in a group expression using the bindings.

binding ::=
  assign (*local_variables*) **with**
    [IDENTIFIER ":="] *expression*

Thus, a binding has a list of variable declarations, optionally initializing one of those declared variables using an expression.

Call-State Expressions

call_state_expression ::=
  "@" *expression*

A call-state expression in the behavior specification of a function is one which is evaluated before a call to that function. Naturally, it can not be applied to expressions like **return** since they have meaning only after a call to the function and using it with a value parameter has no effect. The type of a call-state expression @e is the same type as that of e.

Logical Expressions

Even though C++ has logical expressions, they are short-circuited. So, ADL has special syntax for logical operations that are not short-circuited.

logical_expression ::=
  expression **and** *expression*
  | *expression* **or** *expression*
  | *expression* "==>" *expression*
  | *expression* "<==" *expression*
  | *expression* "<==>" *expression*
  | *expression* "<:>" *expression*
  | **if** "(" *expression* ")" *group_expression*
    (**elsif** "(" *expression* ")"
      group_expression) *
    { **else** *group_expression* } ";"

A logical expression is of boolean type and requires operands of boolean type.

The **and** and **or** operators have their usual logical meaning. The "==>" operator is the logical implication operator, the "<==" operator is the "implied by" operator and the "<==>" operator is the logical equivalence operator.

An expression like

if (c_1) $\{e_1;\}$
elsif (c_2) $\{e_2;\}$
...
elsif (c_n) $\{e_n;\}$
else $\{e;\}$

means that e_k should evaluate to *true* if k is the smallest integer such that c_k is *true* and if no such k exists then e should evaluate to *true* (if the **else** clause is present).

The "<:>" operator is the exception operator and it is used to relate the abnormal behavior definitions and exceptions thrown. Formally, the meaning of a <:> b is

a ==> **abnormal** &&
abnormal && b ==> a.

This says that if the left operand evaluates to *true*, then it denotes abnormal termination. It also says that in case of abnormal termination, if the right operand evaluates to *true*, then the left operand must also evaluate to *true*.

Bounded Quantification

ADL/C++ supports universal and existential quantifiers over finite domains.

quantified_expression ::=
  (**forall** | **exists**) "(" *domain_list* ")"
    group_expression ";"

domain_list ::=
  domain ("," *domain*) *

domain ::=
  variable_declaration ":" *expression*

Here, **forall** and **exists** denotes universal quantification and existential quantification respectively. A quantified expression consists of a variable declaration, an expression that defines a finite domain (currently only integer ranges are allowed) and a group expression. The type of a quantified expression is boolean.

Thrown Expressions

This is used to check if a particular exception is thrown by the function call.

thrown_expression ::=
  thrown "(" *exception_names* ")"

exception_names ::=
  IDENTIFIER ("," IDENTIFIER) *
 | "..."

The type of a **thrown** expression is boolean. Each of the identifiers should be declared as an exception in the exception list for this function.

This expression evaluates to *true* if an exception corresponding to the type of each of the identifiers is thrown. Note that multiple exceptions cannot be thrown by a C++ function. However, because of subclassing, a derived class exception can also be considered as an exception of a superclass type. If "..." is used, then the expression evaluates to *true* if there is some exception thrown.

This expression is commonly used with the exception operator to specify what exception can be thrown under what conditions in case of abnormal behavior.

Unchanged Expressions

This can be used to specify expressions whose values are the same before and after a call to the function.

unchanged_expression ::=
  unchanged "(" *expression_list* ")"

expression_list ::=
  expression ("," *expression*) *

The type of an **unchanged** expression is boolean. It evaluates to *true* only if the values of all the expressions listed remain unchanged before and after a call to the function in whose semantics clause the expression appears.

Other ADL Expressions

The other ADL keywords **return**, **normal** and **abnormal** denote the return value of the function call, and the predicates corresponding to normal and abnormal behavior for that particular function respectively. Since these are properties of the termination of the function, they can not be used in the call-state.

4.4 Preprocessing

A header file consisting of function and class declarations can be included using the standard C++ preprocessor directive ("*#include*") in an ADL/C++ specification file for giving annotations. This is analogous to the way function implementations are developed in C++. To facilitate this, an ADL/C++ specification is preprocessed using some C++ preprocessor before it is processed by any of the ADL tools.

5 Reusing Specifications

Sometimes the behavior (post-condition) of a function might just be a simple refinement of an existing function (behavior). But, ADL does not provide a way to specify this refinement directly. In stead, there is a more general construct to use the semantics (behavior specification) given in ADL/C++ of a function while giving the semantics of other functions.

behavior_expression ::=
  behavior "(" { *selector_expr_prefix* }
    IDENTIFIER < { *param_list* }
    { : *expression* } >

selector_expr_prefix ::=
  expression ("." | "->")

The behavior operator requires a function name and values for parameters and a value to denote the return value of a call to the function with those values. The actual parameters might also include (a pointer to) the object in the case of member functions. The syntax is very close to a function call, so for members, the object (or pointer) is supplied using the appropriate member selector operator.

In "*behavior*$(a.f)$<$arg_1, ..., arg_n$: $retVal$>", the types of the expressions $arg_1, ..., arg_n$ obey all the rules that the types of actual parameters to f do and the type of *retVal* should be the return type of f. The type of a behavior expression is boolean.

"*behavior*$(a.f)$<$arg_1, ..., arg_n$: *retVal*>" evaluates to *true* iff the semantics clause corresponding to the (member) function *f* of *a* does, when evaluated in the scope in which it is specified, by setting the values of its n parameters to $arg_1, ..., arg_n$ respectively, and setting the value of **return** to *retVal*.

Consider the following specification of a *Sort* function :

```
specification SortSpec
{
   typedef Record RecordArray[100];

   RecordArray Sort(RecordArray records)
   semantics {
      // First say it is a permutation
      IsPermutation(return, records);

      // Then say that the return value
      //  has keys in nondescending order
      forall (int i : int_range(0,
                       size(records) - 2))
      {
         return[i].key <=
                    return[i + 1].key;
      };
   };
};
```

Now consider a stable sort function, *i.e.*, a sort function which preserves the relative order of elements with equal keys after sorting. A specification for this function can be written as a refinement of the sort function to say that it sorts and also preserves the order of elements with equal keys. In this case, the specification *StableSort* will be concise and intuitive if it can be written using the existing postcondition for *Sort* in the spirit of reuse supported by C++ by way of inheritance. This can be done as follows :

```
with [SortSpec]
specification NewSortSpec
{
   RecordArray StableSort(RecordArray
                                records)
   semantics {
      // It behaves like Sort.
      behavior(Sort) <records : return>;

      // Relative order of elements
      // with equal  key is preserved.
      forall (int i, j : int_range(0,
                    size(records) - 1))
      {
         if (i < j && records[i].key ==
                    records[j].key)
         {
            exists (int m, n : int_range(
                  0, size(records) - 1))
            {
               m < n &&
               records[i] == return[m] &&
               records[j] == return[n];
            };
         };
      };
   };
};
```

As the expression language of ADL/C++ includes C++ expression language, a function call in a specification refers to its implementation, not the specification. Since some function implementations can have side-effects, it may not be correct to invoke function implementations. In those cases, behavior expression can be very useful.

Another situation where this will be useful is when one class contains objects of some other class and uses the contained class methods to implement its own methods. Then the specifications cab also be written using the specifications of the contained objects using the behavior expression. This gives modularity to specifications similar to the modularity of implementations as supported by C++.

For example, consider a bank class which has methods to deposit to and withdraw from an account given the account number. Using the previously specified *Account* class, a specification of the bank's deposit method can be given as :

```
with [AccountSpec]
specification Bank {
   class Bank {
      Deposit(int acNum, long amt)
         throw(int acNotFound)
      semantics {
         assign Account *acPtr with
            acPtr = GetAccount(acNum);
         if (normal) {
            behavior(
               acPtr->Deposit)<amt>;
         };
      };
   }
};
```

This specification just says that the bank's deposit method merely locates the account corresponding to the given account number and then the amount is deposited into the located account. It also throws an exception if it can not find an account with a given number.

An advantage with this specification is that if later on some subtypes are added to the *Account* class, say something like an account with overdraft protection, the bank specification need not be rewritten, much like C++ implementation of the *Bank* class need not be rewritten (if it is properly implemented).

6 Testing and Validation

One of the important benefits of writing ADL/C++ specifications is automatic test-validation. A C++ implementation can be tested by giving it a sample input data and then the test can be validated to see if it is a success or failure using the ADL/C++ specification.

At present, only unit testing of (member) functions is supported. Given a function declaration to be tested along with a sample data, the validation can be done in the following steps.

- Store the values of all the parameters that are referred to in the semantics part of the function's behavior specification.
- Evaluate all the call-state expressions using the given sample input data and store them in temporary variables.
- Invoke the function with the given sample data to get any return values.
- Evaluate the predicates for normal and abnormal. If neither one evaluates to *true*, then the test *failed*.
- Using the stored parameter values, call-state values and the return value, evaluate each of the expressions that is given in the behavior specifications.
- If any of the expressions evaluates to *false*, then the test *failed*, that is, the (C++) implementation does not satisfy the ADL/C++ specification for that particular input data. Otherwise, the test *succeeded*.

Intuitively this means that any test should conform to either normal or abnormal behavior, if not there is something wrong with the implementation, that is, it is either returning a value or throwing an exception that it is not supposed to. In addition, a test might fail because it did not cause the appropriate state change that it is supposed to cause.

For example, consider the following (wrong) implementation of the *Account* class whose specification is given in Figure 2.

```
#include "account.hh"

void Account::Deposit(long amount)
     throw(Account::NegativeAmount na)
{
   balance = balance + i;
}

void Account::Withdraw(long i) throw
     (Account::NegativeAmount na,
      Account::InsufficientFunds isf)
{
   if (i >= balance)
      throw Account::NegativeAmount;

   if (i >= balance)
      throw Account::InsufficientFunds;

   balance == balance - i;
}
```

As it can be seen, the *Account::Withdraw* method is wrong because it does not update the balance. This is not an unlikely error because of the lexical similarity between the C++ operators == and =. Similarly the *Account::Deposit* method is wrong because it does not check for the value of *amount* being negative in which case it is supposed to throw and exception.

Now, any test for the *Withdraw* method with a nonnegative amount will make its post-condition false, and thus can be classified as a failure. Even though this may be a simple bug to catch, there might be other examples where there might be hard-to-catch bugs that can be detected by testing with appropriate test data (and a correct specification). Similarly, a test of *Account::Deposit* with a negative value for *amount* will fail because the first assertion in the semantics fails because *amount* is negative and the function did not terminate abnormally (as the *NegativeAmount* exception is not thrown).

Thus, an ADL/C++ specification gives some kind of formal basis for test validation rather than the tester deciding whether the test failed or succeeded based on some informal specification. It is also possible to give a more fine-grained test report in case of a failure, by looking at the individual predicates that have failed for a particular test data.

Since ADL/C++ is very close to C++, it is relatively easy to generate code from ADL/C++ specifications that can used for automating the test validation process. This has the advantage that a large number of tests can be run automatically whenever an implementation is modified. Detailed reports for failed tests can also be generated which can help in quickly locating the problem in the implementation that caused the test failures.

When testing using ADL/C++ specifications, there could be calls to constructors, assignment operators and copy constructors. Hence, users should keep these relatively simple and clean, otherwise, the testing might become inefficient and imprecise.

At present, the test data has to be supplied by the user. We are currently working on coming up with a language like TDD [8] to specify test data for testing C++ member functions. In fact, to a large extent, TDD itself can be used to specify test data for C++ functions as well.

We are also looking at ways to generate "interesting" test data from ADL/C++ specifications. For simple data types, it seems to be very feasible to automatically generate reasonable test data. For example, in the *Account::Deposit* method, a negative value for the parameter is interesting because it can cause an exception to be thrown. So, any reasonable test data should have at least one such value for *amount*.

7 Comparison and Related Work

A++ [1] is a language for annotating C++ programs with semantics constraints. This is close to ADL in spirit in that it uses (mostly) C++ syntax for doing this. But, unlike ADL, A++ provides facilities for writing complex specifications, like legal values for datatypes, axioms defining classes i*etc*. Developing any practically usable tools using A++ is ambitious, because specifications could range from high-level axioms for classes to low-level assertions for statements. On the other hand, we have started with the modest goal of associating behavior specifications to functions. This is shown to be achievable and very useful by the success of ADLT using ADL/C. So, have we adopted a similar framework for C++ and are building on top of this to incrementally support specification of other features of C++, like classes and inheritance. It has been demonstrated by our semantic constraints on non-static virtual functions, that it is possible to develop ADL/C++ for specifying the more complex features of C++ with this simple idea as the basis. The A++ annotations are *intrusive*, *i.e.*, embedded within programs, making it difficult to use with pre-compiled libraries. ADL/C++ specifications are *non-intrusive* and hence can be used for specification even when the source code is not available.

The other effort with similar goals that we know of is Larch/C++ [5]. This is a member of the Larch family of languages. This supports two-tiered specifications. First the abstract values are described in the larch shared language LSL. Then the specifications of classes are given in Hoare-style pre and post-conditions for member functions. In our opinion, this is a burdensome process especially for practitioners who want to use formal specifications, because specifications have to be written in a different language with totally different semantics. In ADL also the language is different, but it is built around C++ and we made it a point to introduce as little new notation as possible. We should also point out that Larch has a large collection of LSL traits which can be used to describe most commonly occurring data structures. However, to be able to pick the correct trait, one needs to understand the semantics that describe the behavior of the trait.

To the best of our knowledge, there is no support for testing C++ implementations against Larch/C++ specifications. Larch has a very good collection of tools, most important of which is LP, the larch prover. One can hope to do some reasoning about Larch/C++ specifications using LP. At present there is no support for reasoning about ADL specifications.

8 Conclusions and Future Work

We have designed a language ADL/C++ to write behavioral specifications for C++ programs. The important aspects of this language are :

- *Is easy to learn and use for real software engineers.* We have been careful in designing specification constructs that are simple and easy to use for the real software engineer. While this has reduced the expressivity of the language to some extent, it is surprising how much can still be expressed. ADL/C has shown itself to be useful and practical at the same time, and ADL/C++ will only improve on this.

- *Specifications can be compiled into checking code.* All ADL/C++ specifications may be compiled into code that can verify correct behavior (or conformance) of API implementations. Note that this is different from "executable specifications" where it is possible to generate executable implementations directly from a specification. In general executable specifications impose undue restrictions on specification constructs to simplify the process of generating executable code. ADL/C++ specifications offer a reasonable middle ground where the specifications may be used to efficiently check conformance, while at the same time allows easy development of specifications.

- *Specifications may be developed independently of the program.* ADL/C++ specifications may be written as separate files (that "include" C++ header files). Hence ADL/C++ facilitates the development and use of specifications by organizations that do not have access to C++ source code (*e.g.*, some proprietary API's come with precompiled libraries).

- *Various kinds of tool support is possible.* Checkable specifications may be used in many ways. A system such as ADLT uses ADL/C++ for conformance testing. In such a system, test-data is produced independently of the writing of the specifications. The tool then generates a driver that exercises the API on the test-data, checking the system behavior each time with the specifications. Specifications may also be used to generate self-checking interfaces in cases where security is an issue. Self-checking interfaces are also useful during the development cycle.

The design of the language is not yet complete. We merely extended the ADL/C-style post-condition specifications for functions to specify class method behaviors with some obvious semantic constraints for virtual member functions.

In C++, classes can also be considered as the basic programming units in addition to global functions like in C. So, it would be nice if class behaviors can be specified and *tested* just like function behaviors.

Specification of (methods of) abstract classes is difficult at present. This is because, usually abstract classes do not have any implementation (or state) information that can be used for the post-conditions. So, the only way to do this at present in ADL is to come up with some crude implementation (using the auxiliary declarations) and writing the post-conditions using that. This makes specifying abstract classes complicated.

One way to solve these problems is to adopt an algebraic specification framework, but it is not clear how testing can be supported in that framework.

Considering these issues, it appears some kind of trace-based specifications on top of the current ADL/C++ specifications would be a nice way to capture class behaviors, at the same time preserving the two defining features of ADL - testability and closeness to the base implementation language. We have some preliminary ideas in this respect and we will be incorporating them into ADL/C++ soon.

Availability

The development of the tool is currently in progress. The initial version will be available in about a year. Contact the authors for further information.

Acknowledgments

We would like to thank Rajiv Joshi, Ashvin D'Souza, Roongko Doong and Mark Hefner for their participation in the numerous discussions during the design of the language. We would also like to thank Deepak Kapur for his invaluable suggestions for improving the overall quality of the paper.

References

[1] M. Cline and D. Lea. The Behavior of C++ Classes. *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications*, Marist College, 1990.

[2] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design, Inc., and SunSoft , Inc. The Common Object Request Broker : Architecture and Specification. Pages 45-80. OMG Document Number 91.12.1, Revision 1.1. December 1991.

[3] John V. Guttag and James J. Horning with S. J. Garland, K. D. Jones, A. Modet and J. M.

Wing. Larch : Languages and Tools for Formal Specification. Springer-Verlag, 1993.

[4] D. Kapur and D. R. Musser. Tecton: A Framework for Specifying and Verifying Generic System Components. Rensselaer Polytechnic Institute Computer Science Technical Report 92-20, July, 1992.

[5] Gary T. Leavnes. Larch/C++ Reference Manual. Department of Computer Science, Iowa State University, Sept 1995.

[6] Barbara Liskov and Jeannette M. Wing. A Behavioral Notion of Subtyping. *ACM TOPLAS* 16(6):1811-1841, Nov. 1994.

[7] Sriram Sankar and Roger Hayes. ADL: An Interface Language for Specifying and Testing Software. In *Proceedings of the Workshop on Interface Definition Languages*, January 1994.

[8] Sriram Sankar, Roger Hayes. Specifying and Testing Software Components using ADL. SMLI TR-94-23. *Sun Microsystems Laboratories, Inc.*, April 1994.

[9] J. M. Spivey. Understanding Z, A Specification Langu ge and its Formal Semantics. Cambridge University Press, 1988. Tracts in Theoretical Computer Science, Volume 3.

[10] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 1991.

[11] Sun Microsystems Inc., U.S.A., and Information Technology Promotion Agency, Japan. ADL Translator Design Specification. Document number MITI/0001/D/0.1. August 1993.

Software Composition with Extended Entity-Relationship Diagrams

Pornsiri Muenchaisri and Toshimi Minoura

Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
muenchp@cs.orst.edu
minoura@cs.orst.edu

Abstract

We introduce a compositional approach to application software construction. In our approach, an *extended entity-relationship diagram* (EERD), which represents the component types and the relationship types within an application domain, is used as a template of executable programs in that application domain. As we use *structural active objects* as the components of a program, we can obtain an executable program simply by connecting them as dictated by an EERD. Furthermore, the graphical editor in the proposed software development environment uses EERDs as menus in constructing application software. An EERD used as a menu can enforce legitimate patterns of relationships among components, besides that they provide an intuitive view of available components and possible relationships among them.

1 Introduction

In the late 1960s practitioners and researchers began to discuss a software crisis in developing a large software system with existing methods [Pree87, Gold95, Lamb87, Lewi91, Somm92]. In order to improve productivity and quality in software development, many new techniques and methods to manage complexity of software were developed, but the problem has remained unresolved [Booc94], and serious research still continues in this area.

The *entity-relationship* (ER) approach was first proposed by Chen [Chen76]. Since then it has been extensively used in designing schemas for database systems [Elma89] and to represent the structures of systems in systems analysis [Chen83]. An ER diagram not only provides an intuitive view of an application, but it is also possible to generate automatically a relational schema from it [Teor82]. The ER approach turned out to be very effective in the areas of data management and systems analysis.

In the area of general software development, the object-oriented (OO) approach was one of the most successful [Cox86, Gold95]. To represent the structures of OO software, *Object Modeling Technique* (OMT) [Rumb91], the *Booch method* [Booc94], and the *Fusion method* [Cole94] were developed. Although the object models are extensions of ER diagrams, they incorporate various additions including behavior modeling. Also skeleton code can be generated from an object model.

Composing application software from software components, as other industrial products are produced from their components, has been an aim of many researchers [Cox86, Budd91, Nier91, Jaco92, Mino93b, Seli94, Shaw95b]. By using well-tested software components, we can reduce the development time and enhance the quality of application software. Cox presented the notion of *Software-ICs*, which are reusable software components produced from components from other applications [Cox86]. Nierstrasz et al. developed a *visual scripting* tool called Vista [Nier91, Nier92]. With this tool, a user can interactively construct an application from pre-packaged, plug-compatible software components by direct manipulation and graphical editing.

The *structural active-object system* (SAOS) approach was an attempt to generate executable programs from their graphical representations [Mino93a, Mino93b, Mino93d]. To make a program composed from interconnected components executable, composable active objects, called *structural active objects*, are used as building blocks of software. Structural active objects encapsulate not only data but also control and are better modularized than ordinary objects. Structural active-objects are interconnected with each other through their *structural interfaces*, and composite active-objects thus constructed are hierarchically composed. This mechanism of component composition is referred to as *structural and hierarchical object composition* (SHOC).

Schappert, Sommerlad, and Pree proposed software components accompanied by an *active cookbook* [Scha95]. An active cookbook provides recipe-like on-line guides to help the user compose working software from the framework components. How components can be composed is determined by the *relations* defined among them.

In this paper, we present the design of a software development environment, called *Entity-Relationship Software Development Environment* (ERSDE), for creating executable software automatically. The environment is based on the ER approach with structural active objects. The environment uses an *extended entity-relationship diagram* (EERD) (*domain-specific schema*) as a menu for a graphical editor as well as as a template of executable programs. A programmer can see in a EERD the available entity types and the patterns of the relationships (possible connections) among them. She can compose an application software by instantiating entities from the entity types in the EERD and then by connecting them following the patterns of relationships specified in the EERD. Since entities are implemented as active-objects, the application is executable as soon as entities are instantiated and interconnected.

In Chapter 2, we provides an overview of our approach by using a simple example. Chapter 3 gives a description of ERSDE. In Chapter 4, we demonstrate effectiveness of our approach by applying it to several application domains. Chapter 5 concludes this paper.

2 Overview

In this section we give an overview of our approach with a simple example. Fig. 1 shows a tank system consisting of tanks, valves, and pumps. A tank contains some kind of liquid, a pump makes liquid flow, and a valve controls the amount of liquid that flows through it. The liquid flows from left to right from tanks to other tanks through valves and pumps.

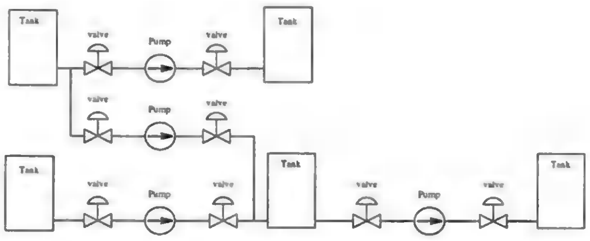


Figure 1: A tank system.

Fig. 2 is a conventional entity-relationship (ER) diagram for tank systems consisting of tanks, valves, and pumps. The input end of a pump can be connected to the output end of a valve, and the output end of a pump to the input end of a valve. The input end of a tank can be connected to the output ends of many valves, and the output end of a tank to the input ends of many valves.

A simple way to enhance understandability of an ER diagram is to replace the rectangular representation of entity types by their iconic

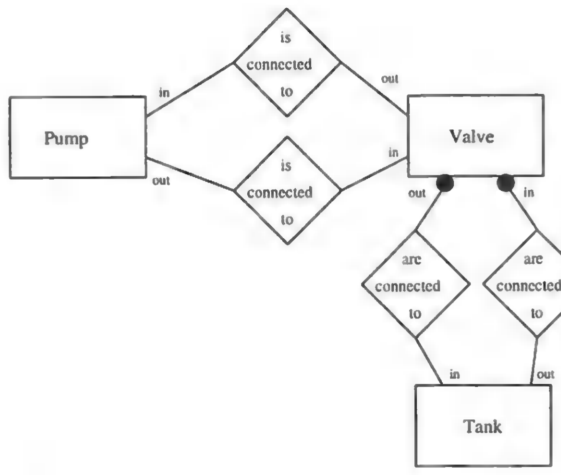


Figure 2: A conventional entity-relationship diagram for tank systems.

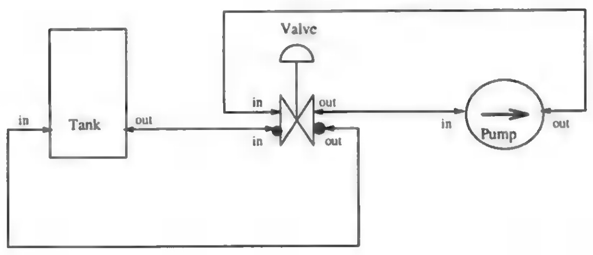


Figure 3: Extended entity-relationship diagram with iconic entity types.

representations. An iconic entity type looks like a real entity. For example, a picture of a valve can be used to represent an entity type `Valve`. Intuitiveness of a domain-specific schema is enhanced with iconic entity types. Fig. 3 shows an *extended entity-relationship diagram* (EERD) of the tank system represented in this way. In this diagram, relationship types are represented by arrows.

However, the representation as shown in Fig. 3 has the following problem. The arrow between the output end of the valve type and the input end of the pump type and the arrow between the output end of the pump type and the input end of the valve type may mislead us to believe that there is a circular connection between a valve and a pump. Similarly, it looks like there is a circular connection between a tank and a valve. This problem will make the diagram confusing or at least unattractive.

To solve the problem described above, we introduce *proxy entity types.* A proxy entity type, which are drawn with dashed lines, is equivalent to the original entity type, and all the connections made to it have the same effect as they were made to the original one. Fig. 4 shows the EERD for tank systems with a proxy of the valve type and a proxy of the tank type. In this way, we can eliminate circular connections which may not exist at the entity level.

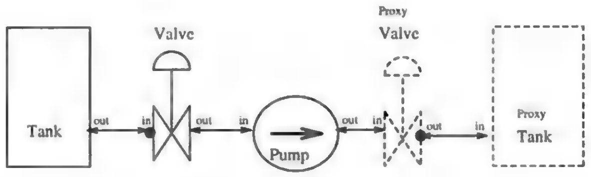


Figure 4: Extended entity-relationship diagram with proxy of entity types.

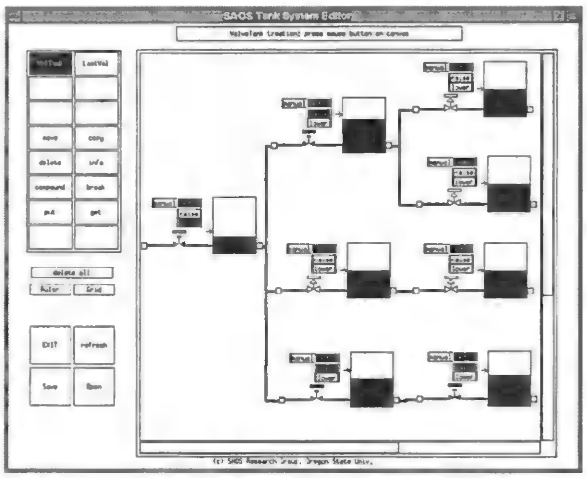


Figure 5: An application of a tank system.

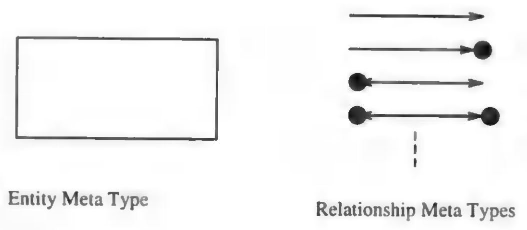


Figure 7: Simplified meta schema.

3 Entity-Relationship Software Development Environment

In this section we present a general framework of a software development environment based on our approach. As shown in Fig. 6, the environment, which we call ERSDE (Entity-Relationship Software Development Environment), consists of three major parts: the *meta schema*, the *schema editor*, and the *application editor*. The major characteristic of ERSDE is that it can be used to develop executable application software in different application domains by component composition.

The meta schema, which contains the *entity metatype* and the *relationship metatypes* as shown in Fig. 7, is used as a template for creating *domain-specific schemas*. When an entity type is created from the entity metatype, the rectangular representation of an entity type may be replaced by an *iconic* representation of the entity type. The relationship metatypes indicate the *cardinality ratios* (*one-to-one*, *one-to-many*, *many-to-one*, *many-to-many*) and the *directions of access*. Additional notations used by the meta schema are described in the next subsection.

A domain-specific (application-specific) schema is an EERD consisting of entity types and relationship types among them. We have already shown an example of an EERD for tank systems in Fig. 4. The schema editor is used to construct domain-specific schemas by instantiating the entity metatype and the relationship metatypes in the meta schema. The schema editor can create and modify EERDs in different application domains.

We can construct applications (instance diagrams) in each application domain with the application editor. This application editor uses the EERD in each application domain as an editor menu. The application editor allows applications to be composed in different application domains by switching the EERD used as its menu.

A SAOS application editor can create executable programs by component composition. However, each SAOS editor is domain-specific, and its menu is a list of items, not an EERD. The SAOS application editor for tank systems is shown in Fig. 5.

In the next three subsections, we describe more details of the three major parts of ERSDE.

3.1 Meta Schema

The meta schema provides notations for creating entity types and relationship types for domain-specific schemas. We adopt some conventional notations and propose some new ones for the meta schema. The meta schema shown in Fig. 8 provides notations for the *entity metatype*, *entity subclassing*, *entity composition*, *relationship metatypes*, and *proxy entity metatype*. The first four notations are extensively used in many object-oriented design methods including OMT, Booch, Fusion [Rumb91, Booc94, Cole94]. However, in our approach, the arrows representing relationship types indicate the directions of data access. A visibility graph of the Fusion method uses an arrow to indicate the direction of data access as we do. However, only one way of data referencing is allowed [Cole94]. The concepts of proxy entity types and relationship representation by proximity, which we describe later, are new.

The entity metatype is the type for the entity types in EERDs. An entity type is instantiated from this entity metatype. Although the generic notation for an entity type is a rectangle, it can be replaced by an iconic representation in a domain-specific schema.

Entity subclassing is a mechanism for an en-

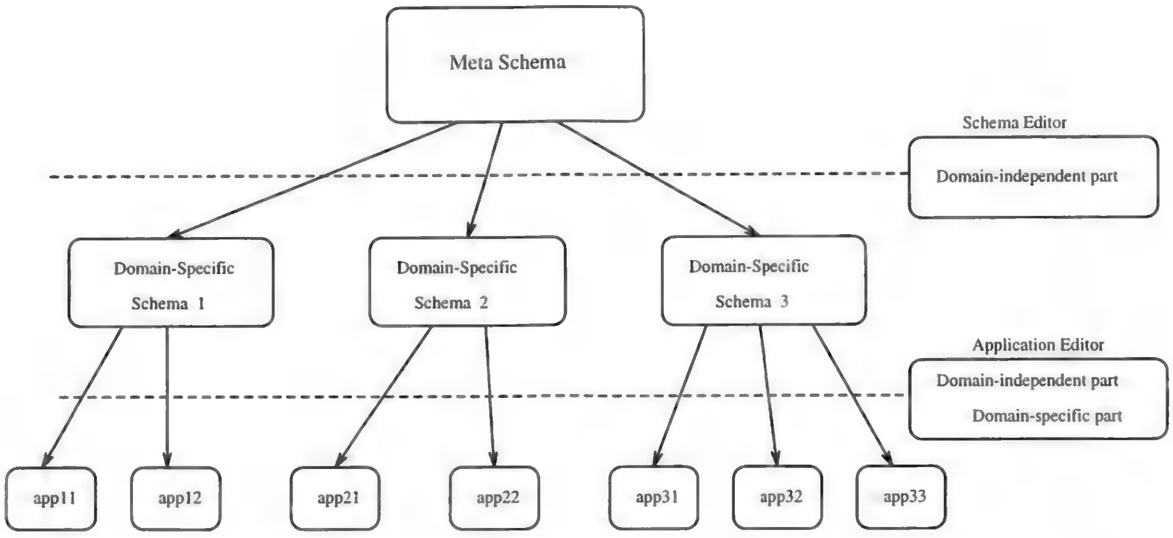


Figure 6: Structure of the ERSDE software development environment.

tity type to inherit some characteristics from another entity type. We use the same notation for entity subclassing as is used by OMT, where entity subclassing is represented by a small triangle in the middle of the line that connects a parent entity type to its child entity types.

Entity composition is a mechanism to allow hierarchical composition of an entity from its component entities. A *composite entity* is an entity created by entity composition. A *composite entity type* is represented by a rectangle with double dashed outlines. Such a rectangle encloses the entity types of its members.

The relationship metatypes in the meta schema are templates for creating relationship types in EERDs, where entity types are connected with other entity types by relationship types. A relationship metatype is a *unidirectional* or *bidirectional* arrow with small filled-circles at its ends. A small filled-circle (•) means "many". The tag attached to a relationship type represents the attributes of the relationship type. The semantic direction of a relationship is normally from left to right or from top to bottom. In this paper, we refer to the source of a relationship as its left-side entity, and the destination as its right-side entity. There are four possible combinations of these small filled-circles.

1. *One-to-one*: The relationship metatype is one-to-one if there is no small filled-circle at either end of the arrow.
2. *One-to-many*: The relationship metatype is one-to-many if there is a small filled-circle at the right end of the arrow.
3. *Many-to-one*: The relationship metatype is many-to-one if there is a small filled-circle at the left end of the arrow.
4. *Many-to-many*: The relationship metatype is many-to-many if there are small filled-circles at both ends.

The direction of an arrow indicates that of data access. The access direction of data may be different from the semantic direction. There are three possible cases for the direction of an arrow.

1. *Right-end access* (E1 $\longrightarrow$ E2): If an arrow head is at the right end of the arrow, an instance of E1 can access an instance of E2, but the instance of E2 cannot access the instance of E1.
2. *Left-end access* (E1 $\longleftarrow$ E2): If an arrow head is at the left end of the arrow, an instance of E2 can access an instance of E1,

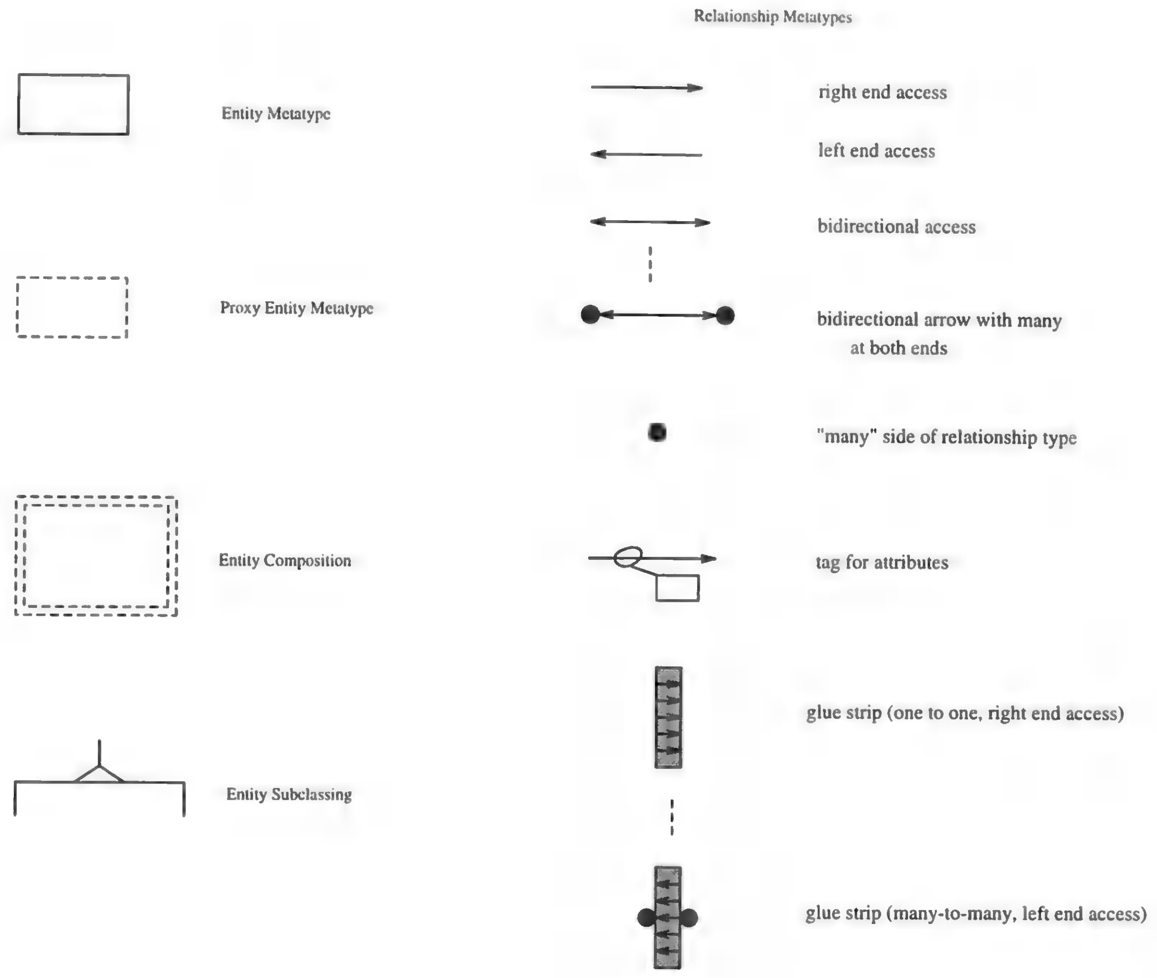


Figure 8: Meta schema.

but the instance of E1 cannot access the instance of E2.

3. *Bidirectional access* (E1 $\longleftrightarrow$ E2): If arrow heads are at both the right and left ends, an instance of E1 can access an instance of E2, and the instance of E2 can access the instance of E1.

All possible combination of cardinality ratios and access directions for the relationship metatypes are given in Fig. 9. Fig. 9(a) shows the one-to-one relationship metatype where an instance of E1 can access an instance of E2, but the instance of E2 cannot access the instance of E1. Fig. 9(c) shows the one-to-many relationship metatype where an instance of E1 can access multiple instances of E2, but these instances of E2 cannot access the instance of E1. Fig. 9(g) shows the many-to-many relationship metatype where an instance of E1 can access multiple instances of E2, but these instances of E2 cannot access the instance of E1. Fig. 9(l) shows the many-to-many relationship metatype where an instance of E1 or E2 can access the related instances of E2 or E1, respectively.

In current object-oriented programming languages, relationships are implemented by pointers. Since pointers cannot carry any attribute information, we must store attribute information of relationships in entities. For a one-to-one relationship type, we can move the attributes of the relationship type to the entity type at either the left or right side of the relationship type. Fig. 9(a), Fig. 9(b), and Fig. 9(i) are examples of one-to-one relationship types whose attributes can be moved to entity type E1 or E2. For a one-to-many or many-to-one relationship type, we can move the attributes of the relationship type to the entity type at the "many" side of the relationship type. Fig. 9(c), Fig. 9(f), Fig. 9(j) are examples of one-to-many relationship types whose attributes can be moved to entity type E2. Fig. 9(d) Fig. 9(e), Fig. 9(k) are examples of many-to-one relationship types whose attributes can be moved to entity type E1. For a many-to-many relationship type, we cannot move the attributes of the relationship type to entity type E1 or E2. Therefore, a many-to-many relationship with attributes must be implemented as an entity. Fig. 9(g), Fig. 9(h) and Fig. 9(l) are examples of many-to-many relationship types.

We now explain the reasons why proxy entity-types are introduced. The proxy entity types are designed to make an EERD easy to understand. Proxy entity types are equivalent to their original entity types. The following problems are examples that proxy entity types can solve.

1. (Multiple Sheet Problem) When a system is large, multiple sheets are needed to show all the required entity types and relationship types. Then there should be a way to refer to entity types in other sheets. From one sheet we can refer to an entity type given on another sheet with a proxy entity type.

2. (Circular Connection Problem) This problem occurs when some entities are connected to other entities of the same type. In this case, a chain of relationship types originates from and ends at the same entity type. We have already illustrated this problem and its solution in Fig. 3 and in Fig. 4, respectively.

3. (Multiple Component Problem) This problem occurs when a composite entity type includes multiple occurrences of one entity type as its components. Fig. 10 shows a standard ER diagram for an entity type `Car` which is a composite type consisting of four occurrences of the entity type `Wheel` and one occurrence of the entity type `Body`. The fact that a car has four wheels is not intuitively represented. The EERD given in Fig. 11, on the other hand, shows the composite entity type `Car` by using proxy entity types.

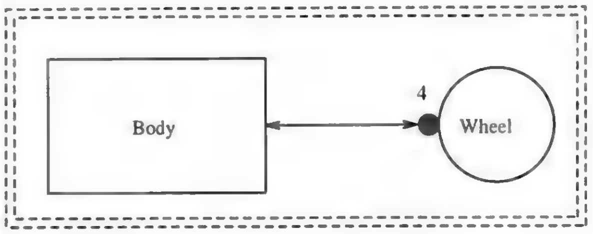


Figure 10: A car as a composite entity type.

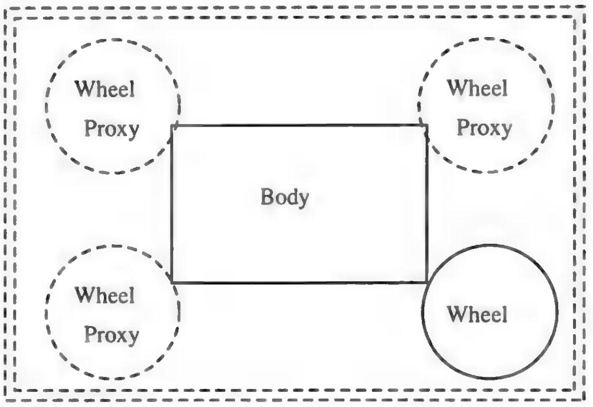


Figure 11: A car as a composite entity type with proxies.

Although the generic notation for a proxy entity metatype is a dashed rectangle, it can be replaced by an iconic representation in a domain-specific schema.

3.2 The Schema Editor

The ERSDE provides a graphical schema editor for creating, deleting, and moving entity types in EERDs. We use the schema editor to build domain-specific schemas. The domain-specific schemas display entity types and relationship types among them.

The schema editor is a (general) domain-independent graphical editor for creating and manipulating graphical representations of entity

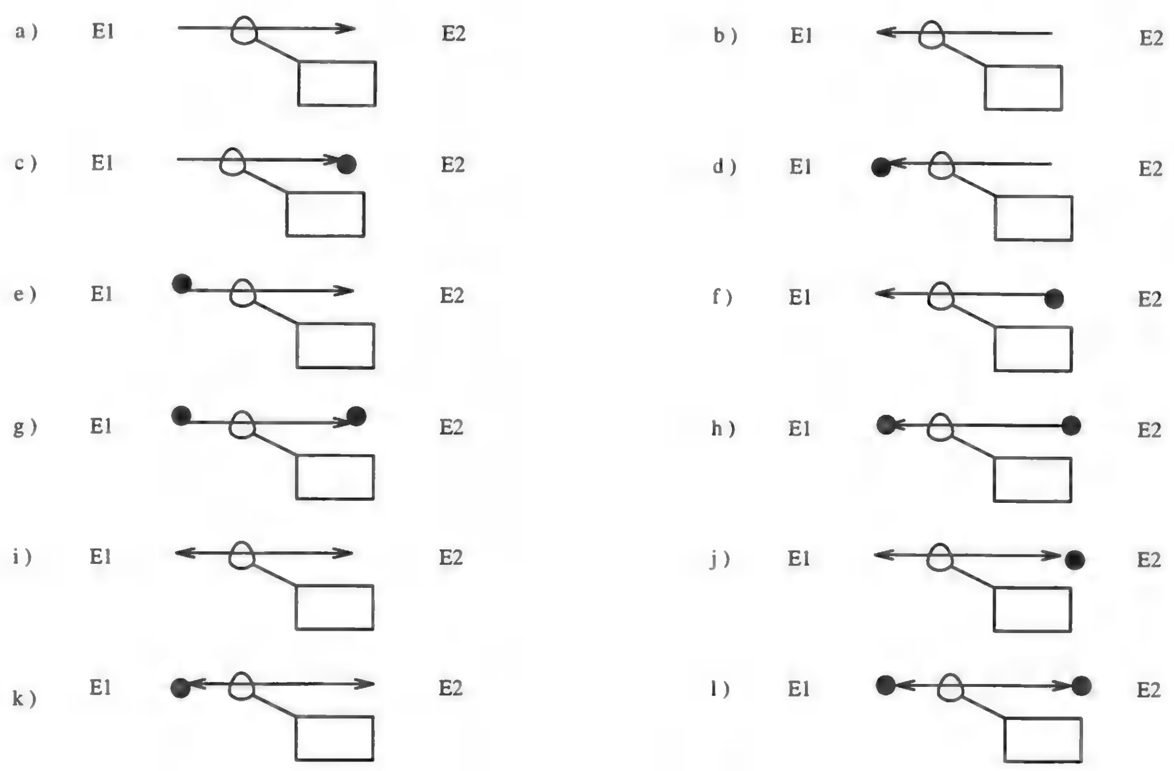


Figure 9: Relationship metatypes between two entity types.

types and relationship types in domain-specific schemas.

There are four major steps in constructing a domain-specific schema. First, we use the schema editor to create entity types. Second, we connect entity types to other entity types by relationship types. Third, once an EERD is completed, the schema editor generates skeleton code for the entity types and the pointer structures to access related entities according to the direction of data access specified in the domain-specific schema. Fourth, a programmer is responsible for providing behaviors for each entity type.

We use some new ideas in domain-specific schemas: iconic representations of entity types, proxy entity types, and *relationship representation by proximity.*

We can relate entity types to other entity types by placing them closely. This mechanism for creating relationship types is called relationship representation by proximity. Although relationship types shown by this mechanism are semantically not different from those represented by arrows, composite (assembly) entity types shown in this way look more like real entities. We use a *grey glue strip* to represent a relationship type by proximity. The cardinal ratio of a relationship type can be indicated with a small filled-circle within an entity type on the "many" side. An EERD for cars using this notation among its component types is shown in Fig. 12. A `Car` has one `Driver` and multiple `Passengers`.

3.3 Application Editor

We can use the application editor to construct applications from a domain-specific schema (EERD). In composing an application, the domain-specific schema is used as the menu of the application editor to instantiate entity types and connect them in compliance with the connectivity styles in the EERD. When used as a menu of a graphical editor, an EERD is more effective than a conventional editor menu since it

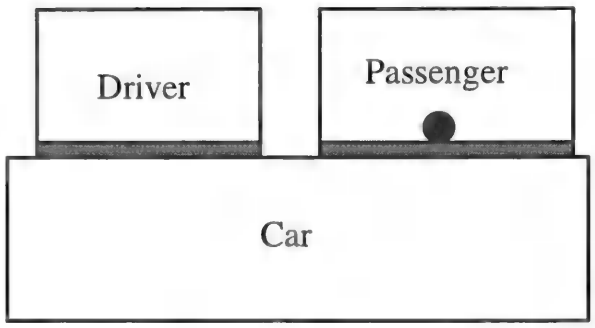


Figure 12: The EERD for a car, a driver and passengers.

can show not only entity types but also possible relatioships between entities.

The application editor consists of the domain-independent part and the domain-dependent part. The domain-independent part is a general graphical editor used to create, and move instances of entity types and connections among them. The domain-dependent part handles application-specific editing operations. For the schema editor, an EERD can be regarded as an instance diagram and the meta schema as a schema.

If each entity is an active object, the application can be executed once the entities are connected. The application allows us to construct applications in different layouts, and to move, delete, and edit components interactively.

4 More Examples

In this section we demonstrate the generality of our approach by applying it to examples in five different domains. For each domain, we provide the conventional ER diagram, the EERD, and an example of application. Our objective is to show that EERDs are more intuitive than conventional ER diagrams.

4.1 Queuing Systems

We first consider queuing systems. Fig. 13 shows a queuing system consisting of a *generator*, three *queues*, and two *processors*. The generator produces jobs, and a queue holds jobs. A processor removes a job from its input queue, processes it, and passes the job to its output queue.

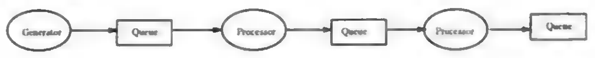


Figure 13: A queuing system.

Fig. 14 is a conventional entity-relationship (ER) diagram for queuing systems. The output end of a generator can be connected to the input end of a queue. The input end of a processor can be connected to the output end of a queue, and the output end of a processor to the input end of a queue. Multiple processors can be connected to a single queue for job input and output.

The EERD for the same application domain is shown in Fig. 15. A proxy of the type `Queue` is used to avoid a circular connection between the type `Processor` and the type `Queue`. The EERD explicitly shows the direction of the data access. An example of a queuing system application instantiated from the EERD is shown in Fig. 16.

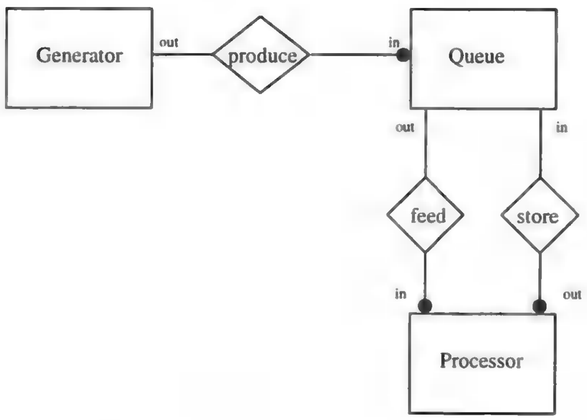


Figure 14: A conventional entity-relationship diagram for queuing systems.

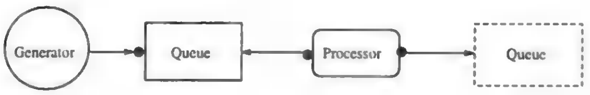


Figure 15: The extended entity-relationship diagram for queuing systems.

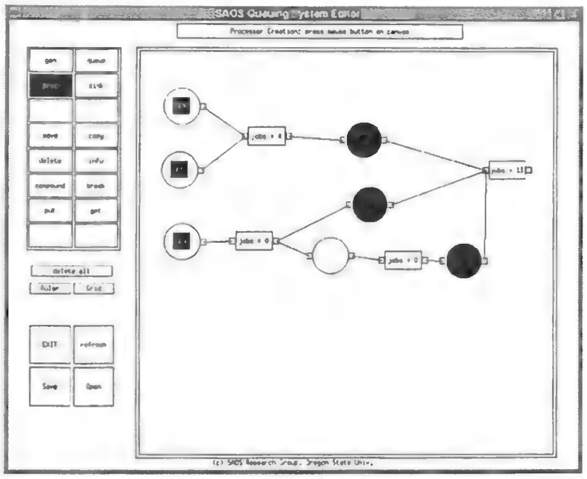

SAOS Queuing System Editor
Processor Creation: press mouse button on canvas
EXIT
refresh
Save
Open
delete all
(c) SAOS Research Group, Oregon State Univ.

Figure 16: A queuing system application.

4.2 Wide Area Network Systems

We now consider simulation of wide area network (WAN) systems. A simulator for a WAN system consists of *hosts*, *links*, and a *recorder*. A host communicates to another host via a link. The load incurred on a link is referred to as *usage*, which we treat as a component of the link. A recorder displays the usage of a link.

Fig. 17 shows a conventional ER diagram for WAN systems. The input end of a link is connected to a host, and the output end of the link is connected to another host. Multiple links can be connected to a host. A recorder is connected to the usage component of a link.

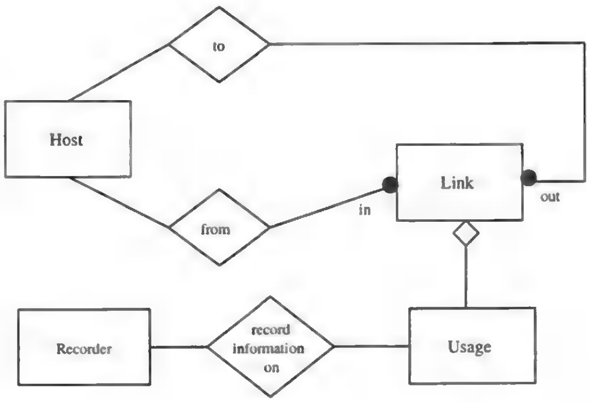


Figure 17: A conventional ER diagram for wide area network systems.

The EERD for WAN systems is shown in Fig. 18. A link is a relationship which has attribute usage. The attribute usage is an entity. A programmer can see how a host can be connected to another host via a link more clearly in the EERD than in the conventional ER diagram. Note that a relationship with attributes must be implemented as an object in a current OO language. An iconic representation is used for type Recorder. Fig. 19 shows an example of a WAN simulator.

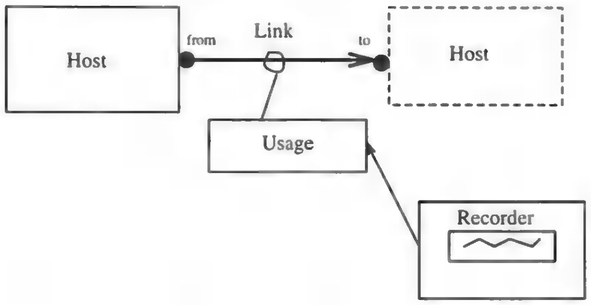


Figure 18: The EERD for wide area network systems.

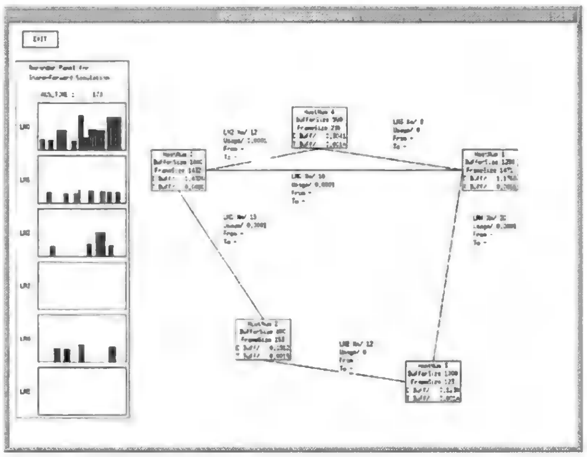


Figure 19: A wide area network system application.

4.3 Local Area Network Systems

Let us now consider simulators of local area networks (LANs). A LAN simulator consists of ethernet *cables*, *stations*, and *repeaters*. A cable is divided into multiple *cable segments*. A station is connected to a cable segment, and a repeater is placed between two cable segments.

The conventional ER diagram for LANs is shown in Fig. 20. A cable consists of multiple cable segments. The left end of a cable segment may be connected to the right end of another cable segment, and its right end to the left end of still another cable segment. One end of a repeater is connected to a cable segment and another end to another cable segment. A station is connected to a cable segment.

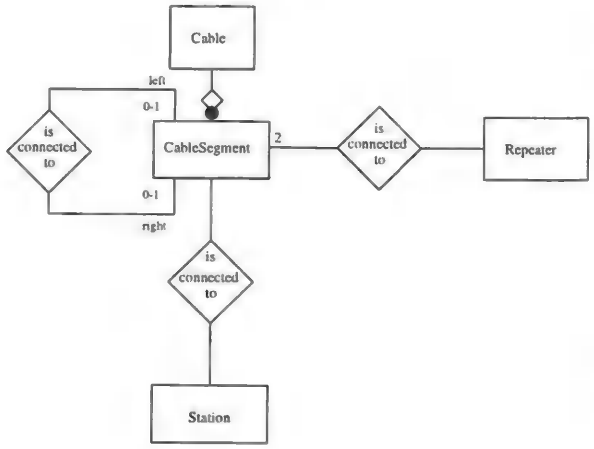


Figure 20: A conventional ER diagram for local area network systems.

The EERD for LANs is shown in Fig. 21. The fact that a cable consists of an array of cable segments is graphically shown with dashes (- - -). Furthermore, a connection of a cable segment to another one at its end is shown by a grey (*glue*) strip representing a relationship. A grey strip is one way to represent a relationship by graphical proximity. Fig. 22 shows an example of a LAN simulator.

4.4 A Digital Circuit Simulator

We now apply our approach to digital circuits. A digital circuit consists of various (AND, OR, NOT, NAND, NOR) *gates*, *signal generators*, *clock generators*, and *recorders*.

Fig. 23 shows the conventional ER diagram for digital circuits. Each of a SignalGen (signal generator), a clock, and a Gate has an attribute output of type Signal. A recorder takes a Signal as its input. A Gate is a 1-Gate, a

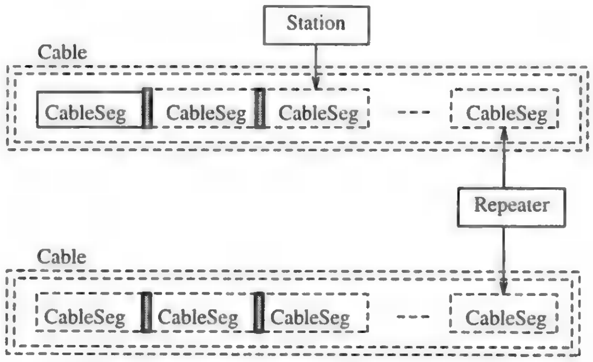


Figure 21: The EERD for local area network systems.

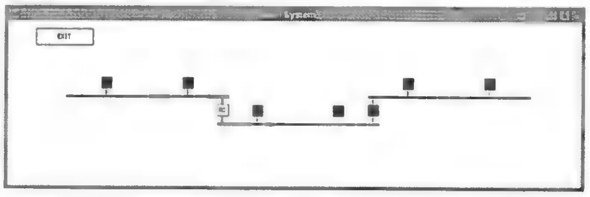


Figure 22: A local area network system application.

2-Gate, or an M-Gate. A 1-Gate takes one input of type Signal, and 2-Gate takes two inputs of type Signal. Type NOT is a subtype of type 1-Gate. We consider AND, OR, NAND, and NOR gates as two-input gates, and we consider M-AND and M-OR gates as multiple-input gates.

The EERD for the digital circuits is shown in Fig. 24. Type Signal Generator, Clock, and Gate are shown as composite entity types. Proxies of type Signal are used as input signals of type Recorder, Type 1-Gate, and type 2-Gate. Type NOT, type AND, type OR, type NAND, type NOR, type M-AND, and type M-OR use their standard iconic representations. Fig. 25 shows a D-latch circuit as an example of a digital circuit.

4.5 Automatic Guided Vehicle Systems

We finally consider simulation of automatic guided vehicle systems (AGVs) used in automated factories. An AGV simulator consists of *track segments*, *intersections*, *stations*, *AGVs*, *jobs*, and *routes*. An AGV travels on the track

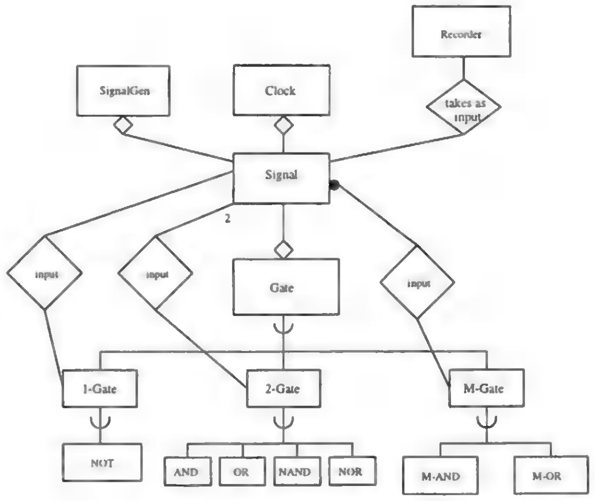


Figure 23: A conventional ER diagram for digital circuits.

segments carrying multiple jobs and following a route. Each job is picked up by an AGV at a station and delivered to another station. An intersection is a connection point of one or more track segments. A track segment must be connected to an intersection at each end. An intersection can also be connected to a station. At any one time, only one AGV can occupy an intersection.

Fig. 26 shows the conventional ER diagram for AGV systems. The diagram shows all possible connections among entity types.

The EERD for AGV systems is shown in Fig. 27. Proxies of `AGV`, `Station`, `Job`, `TrackSeg` (track segment), and `Intersection` are used to make the diagram intuitive. Relationship representations by proximity (grey glue strips) are used between `AGV` and `Station`, `Station` and `Intersection`, `Job` and `AGV`, `AGV` and `TrackSeg`, `AGV` and `Intersection`, `Job` and `Station`, and `Station` and `Intersection`. `AGV` on `Station`, which in turn is on `Intersection`, means that an `AGV` can be at a `Station`, which in turn is adjacent to an `Intersection`. An `AGV` can also be in an `Intersection`. A small-filled circle inside type `Job` on type `AGV` indicates that the relation ships between `Job` and `AGV` are many-to-one, implying that a `AGV` can carry multiple `Jobs`. The left end of a `TrackSeg`

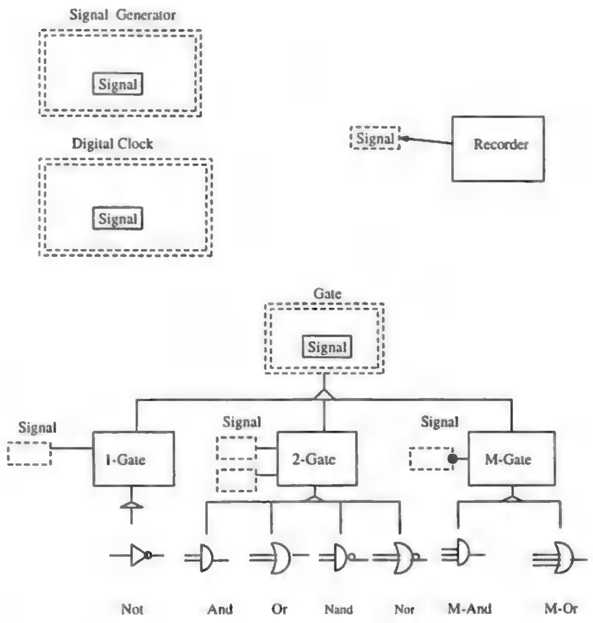


Figure 24: The EERD for digital circuits.

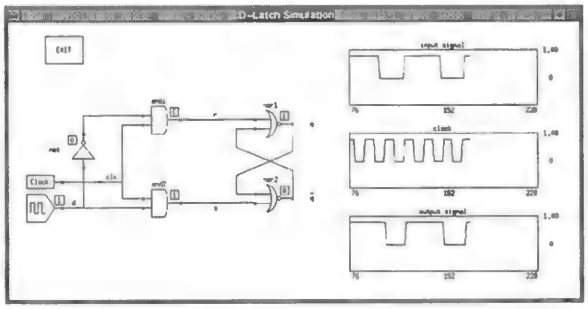


Figure 25: A D-latch circuit.

can be connected to an `Intersection` and the right end of a `TrackSeg` can be connected to another `Intersection`. The fact that multiple `TrackSegs` can be connected to an `Intersection` is shown by small filled-circles within `TrackSeg`. Fig. 28 shows an example of a AGV simulator.

5 Conclusions

We presented a software development environment for composing application software from components. The environment uses *structural active objects* as components, and it includes

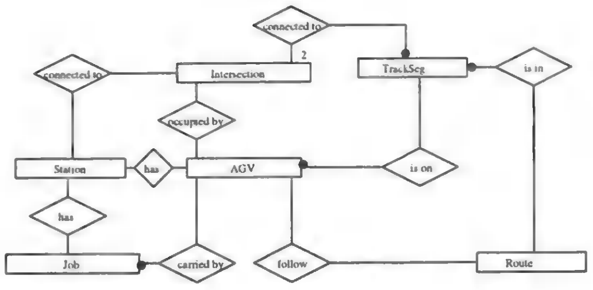


Figure 26: A conventional ER diagram for AGV systems.

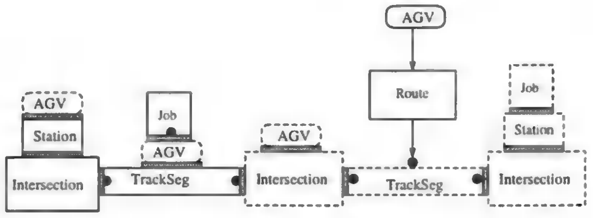


Figure 27: The EERD for auto guided vehicle systems.

three main parts: the *meta schema*, the *schema editor*, and the *application editor*.

The meta schema provides notations for extended entity-relationship diagrams (EERDs). We can use the schema editor to create domain-specific schemas as EERDs by creating entity types and then connecting them with relationship types. Once the EERD for an application domain is complete, the schema editor can generate skeleton code for the entity types and the pointer structures implementing the relationship types.

An application can be composed by instantiating entities from entity types in an EERD and then connecting those entities according to the permissible relationship patterns shown in the domain-specific schema. If behavior definitions have been attached to entity types manually with the schema editor, application software created from the EERD becomes executable.

The ER approach has been used extensively in the data management area and in the system analysis area. We demonstrated that we can construct an executable software program from their components by connecting them exactly

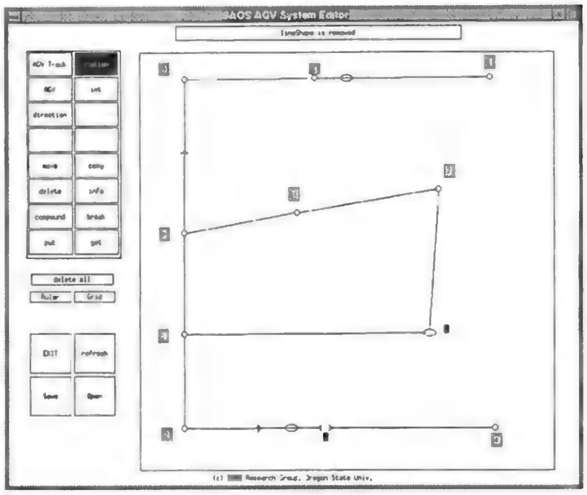

LineShape is removed

Figure 28: An application of auto guided vehicle systems.

as indicated by an extended ER diagram when structural active objects are used as software components.

To use EERDs as templates to compose applications, we introduced some new ideas and notations, e.g., iconic entity types, proxy entity types, and relationship representation by proximity. We then presented the architecture of the entity-relationship software development environment (ERSDE). One new idea is to use an EERD as a menu for a graphical editor. Compared to an ordinary menu, which is simply a list of items that can be created, an EERD can show possible patterns of connections among the entities created. Furthermore, by using iconic representations in the ER diagram, its intuitiveness can be enhanced.

We have implemented several *application-specific* editors that allows us to create application software by component composition. Currently we are developing the schema editor and the application editor so that the application software in different domains can be composed.

References

[Booc94] Grady Booch. *"Object-Oriented Analysis and Design with applications"*, chapter 1, pages 3–26. Benjamin-Cummings, 2 edition, 1994.

[Budd91] Timothy Budd. *"An Introduction to Object-Oriented Programming"*, chapter 1, pages 14–15. Addison-Wesley, 1991.

[Chen76] Peter Pin-Shan Chen. "The Entity-Relationship Model - Toward a Unified View of Data," *ACM TOD* 1(1):9–36 March 1976.

[Chen83] Peter Pin-Shan Chen. "A Preliminary Framework for Entity-Relationship Models," *Entity-Relationship Approach to Information Modeling and Analysis* pages 19–28 1983.

[Cole94] Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, , and Paul Jeremaes. *"Object-Oriented Development: The FUSION method"*, chapter 1-4, 8, 9, pages 1–230. Prentice-Hall, 1 edition, 1994.

[Cox86] Brad J. Cox. *"Object Oriented Programming - An Evolutionary Approach"*. Addison-Wesley, 1986.

[Elma89] Ramey Elmasri and Shamkant B. Navathe. *"Fundamentals of Database Systems"*. The Benjamin/Cummings Publishing Company Inc., 1989.

[Gold95] Adele Goldberg and Kenneth S. Rubin. *"Succeeding with Objects: Decision Frameworks for Project Management"*, chapter 14, page 315. Addison-Wiley Publishing Company, Inc., 1995.

[Jaco92] Ivar Jacobson, Magnus Christerson, Patrik Johnson, and Gunnar Overgaard. *"Object-Oriented Software Engineering: A Use Case Driven Approach"*, chapter 11, page 291. Addison-Wesley, 1992.

[Lamb87] David Alex Lamb. *"Software Engineering: Planning for Change"*, chapter 1, pages 1–6. Prentice Hall, 1987.

[Lewi91] T. G. Lewis. *"CASE: Computer-Aided Software Engineering"*, chapter 1, pages 11–25. Van Nostrand Reinhold, 1991.

[Mino93a] T. Minoura, S. Choi, and R. Robinson. "Structural active-object systems for manufacturing control," *Integrated Computer-Aided Engineering* 1(2):121–136 1993.

[Mino93b] Toshimi Minoura, Shirish S. Pargaonkar, and Kurt Rehfuss. "Structural Active Object Systems for Simulation,". In *OOPSLA'93 proceedings*, pages 338–355. ACM, October 1993.

[Mino93d] T. Minoura and Sungoon Choi. "Structural active-object systems fundamentals,". Technical Report 93-40-04, Dept. of CS, Oregon State University, 1993.

[Nier91] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey, and Marc Stadelmann. "Objects + Scripts = Applications," *In Proceedings, Esprit 1991 Conference* pages 534–552 1991.

[Nier92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. "Components-Oriented Software Development," *Communication of ACM* 35(9):160–165 Sept 1992.

[Pree87] Roger S. Pressman. *"Software Engineering: A Practitionaer's Appproach"*, chapter 1, pages 1–20. McGraw-Hill, 2 edition, 1987.

[Rumb91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *"Object-Oriented Modeling and Design"*, chapter 1-12, pages 1–277. Prentice-Hall, 1 edition, 1991.

[Scha95] Albert Schappert, Peter Sommerlad, and Wolfgang Pree. "Automated Support for Software Development with Frameworks,". In *Symposium on Software Reusability*, pages 123–127. ACM SIGSOFT, April 1995.

[Seli94] Bran Selic, Garth Gullekson, and Paul T. Ward. *"Real-Time Object-Oriented Modeling"*. John Wiley and Sons, Inc., 1994.

[Shaw95b] Mary Shaw, Robert DeLine, Daniel V. Klien, Theodore L. Ross, David M. Young, and Gregory Zelesnik. "Abstractions of Software Architecture and Tools to Support Them," *IEEE Transactions on Software Engineering* 21(4):314–335 April 1995.

[Somm92] Ian Sommerville. *"Software Engineering"*, chapter 1, 11, 16, 17, 18. Addison-Wiley, 4 edition, 1992.

[Teor82] Toby J. Teorey and James P. Fry. *"Design of Database Structures"*. Prentice Hall, 1982.

A Measure of Testing Effort

John D McGregor
Satyaprasad Srinivas
Dept. of Computer Science
Clemson University
Clemson, SC 29634
johnmc@cs.clemson.edu
satya@cs.clemson.edu

Abstract Accurate estimates of the time and resources needed for a project are difficult to achieve. Numerous metrics have been proposed and a few have proved reliable in making these estimates. With the increased emphasis on quality and testing, estimates of the amount of effort required to test a product are a necessary part of any complete project estimate.

Estimates of the effort to test object-oriented components and systems are particularly important because these components are often added to repositories to be used many times. The amount of effort required to test the component is related to its complexity. We consider several measures of method and class complexity and relate them to testability.

The main focus of this research is to estimate the effort that is needed to test a class, as early as possible in the development process. We investigate the testability of a method in a class and indirectly estimate the effort that is needed to test a class. We define a concept termed the visibility component of a method. It is a measure of the accessibility of the information that must be inspected to evaluate the correctness of the execution of a method. We show that the testability of the method is a function of its visiblity component.

Keywords: Testability, Data State Collapse, Syntactic/Semantic complexity.

1 Introduction

The ease with which a software component can be tested is an important element in determining the amount of effort required to build that component. The aggregation of the testing efforts for all of the classes has a major impact on the overall effort required to build a software system. The goals of our metrics research have been expanded from quantifying the construction process by considering the design complexity of individual classes[20][12][1]to quantifying project attributes by using metrics[17]. In this paper we present early results from our work on estimating the amount of effort that will be required to test an object-oriented software system.

Software metrics are quantitative values that provide information for decision making. Metrics are used to quantify attributes of the product being developed and of the process used to construct the product. These measurements can be used to estimate the effort, size and cost requirements for the development of a software system.

Given an iterative, incremental process model, repeated applications of a metric can provide an opportunity to recognize trends that can be used to predict the progress, or lack thereof, of the project. This technique provides an opportunity to identify problems early in the developmental cycle. Project attributes are therefore observed/estimated as early as in the analysis phase, even though there is a risk of reduced accuracy[15].

The main focus of this research has been to estimate, as early as possible in the development process, the effort that is needed to test a class. We have investigated the testability of a method in a class and use that measure to indirectly estimate the effort that is needed to test a class. We introduce a concept termed the *visibility component* of a method. It is a measure of the accessibility of information that must be inspected to evaluate the correctness of the execution of a method. We show that the testability of the method is a function of its *visiblity component.* The estimation of a class's testability given the testability of its methods is also presented. The metric appropriately handles features in object-oriented languages such as the visibility of inherited attributes and other language features such as exception handling.

This paper makes four main contributions:

- This work extends existing research on the testability of procedural software to object-oriented software,
- It considers the levels of information hiding within an object-oriented program, such as protected and private areas, and how these contribute to hiding faults,
- It considers the levels of scoping such as attributes with object or class level scoping, and
- It addresses such modern programming features as exception handling.

The remainder of the paper is organized as follows. In the next section, we review the background and related work necessary to understand the metric. In section 3, we define a set of basic terms used in the description of the metric and define the metric itself. In section 5, we consider our metric in the context of criteria that have been widely used for evaluating complexity metrics and we present an experimental validation of the metric as well. In section 4, we present a set of examples to illustrate the computation of the metric. Finally in section 6, we present conclusions and directions for future work.

2 Related Work

The early work on metrics for object-oriented software includes an OOPSLA paper by Chidamber and Kemerer[5] and a book by Lorenz[13]. The topic of testing metrics for the object oriented paradigm has emerged only as a recent topic. Research in the area of testability metrics is meager and is limited to the procedural paradigm. Several code models/tools have been proposed to determine the testability of procedural code, but these cannot be directly applied to object-oriented programs. These techniques fail to recognize the associations among the methods in a class and they fail to account for the state of the objects created from the class.

Among the measures in the procedural paradigm that are of interest is the Domain Range Ratio metric(DRR) proposed by Voas[24]. This metric is a simple measure that calculates the testability of a procedure based on the information that is flowing into and out of the a procedure. Although DRR establishes an upper bound on the testability of the code, it does not help locate the particular piece of code where an internal data state collapse occurs. Voas has also developed the PIE[23] technique to help identify locations that have a probability of hiding faults. The technique uses a set of semantic and syntactic mutants to help identify locations having a high probability of internal data state collapse[23].

A major goal that has animated our research has been to estimate the effort required for testing a class at a very **early** stage of the development life cycle. Given the fact that the process of testing involves the actual code, early calculation of the metric limits the testing-related attributes that are available for the computation. So our metric calculates an upper bound on the testability of a class. We also feel that the PIE model, although very demonstrative, is computationally intensive and impractical to apply using only specifications. The use of syntactic and semantic mutation testing are also factors influencing our decision.

We have designed the metric to use the design information in the class specifications. The attributes that are available from the specifications include the data attributes of the class and the method names along with their respective signatures. The objects that are returned from each of the methods are also considered.

The metric focuses on the information content in the class specification rather than on the actual interaction of the methods and their messages. Although one might argue that this limits the accuracy of the forecast, it is our opinion that the metric strongly represents areas in the class deserving additional attention. This can be used to estimate and schedule the resource allocation during the testing phase of the lifecycle.

We have *validated* the metric against the formal list of desiderata proposed by Weyuker[25]. Although the list has its drawbacks, the discussion provides one form of theoretical validation of the metric. Additional validation work has been carried out and will be discussed in section 5.

We have also developed a testing architecture that improves the testability of the software by improving the visibility of the internal attributes of the objects[19]. This organizing principle reduces the amount of effort required to test by requiring a test class for each production component and by using language mechanisms to overcome the information hiding built into the design.

3 Metric Definition

The life of a program is a series of *invocations*[3]. During each invocation, the program interacts with the rest of the world, receiving inputs and, as a result, producing outputs. The specification of a program can be used to unambiguously decide whether the program's output is correct, given the input that the program received.

A program may behave non-deterministically, i.e. its results may differ between executions with identical inputs. A program is said to be **correct** on an input i, iff the output produced on that input is compliant with the specification of that program. The program is categorized as being **faulty** otherwise. The set of inputs that cause failures forms the failure Domain *DF* which is a subset of the input domain D[24].

Testability is the prediction of a method's ability to reveal faults in its implementation given a particular input distribution[3]. Testability is dependent on the structure/specification and the input profile of the method. At a very early stage of the developmental cycle, the possible set of inputs and outputs to the method, together with the input profile can be used to determine the testability of a method.

A method specification that maps many inputs to a single output indicates some *information loss* during execution. This indicates that more information flowed into the method than flowed out. A one-to-one specification indicates that no information loss has occured. Unlike procedural systems, a method in an object-oriented program may actually have more information flow out than flowed in because it may access an attribute and return it or some subset of it. Methods that implement many-to-one specifications may produce information that is stored in its internal states and that may affect the results of program execution much later in the computation. It is our hypothesis that this uncertainty affects the amount of testing that must be performed in order to achieve a specified level of test coverage.

3.1 Terminology

Generally, the terminology used to define and discuss the proposed metric is the same as used by the C++ programming community to discuss object-oriented concepts. One exception to this is the use of "method" as opposed to "member function". The decision to employ C++ terminology does not indicate that the metrics are applicable only to C++ designs. Rather it is indicative of the fact that C++ provides a sufficiently broad range of constructs to support discussion of other models[1].

3.2 Basic Terms

- **Explicit parameter** - An object that is explicitly declared in the method signature, see Figure 1.
- **Implict parameter** - An object that is visible to the method by virtue of the fact that it is declared as a class or instance variable in the class declaration. Not all of the data attributes in the class are visible to all the methods in the class. Syntactically all the protected and the public attributes are visible to a derived class during inheritance. Therefore the implicit parameters for a derived class includes the attributes of the base class that are visible to the derived class. A detailed set of counting rules for parameters are presented in section 3.7.
- **Constant Object** - A constant object is one that does **not** change its state during the execution of the method. These consist of objects that are made state invariant syntactically by definition, such as

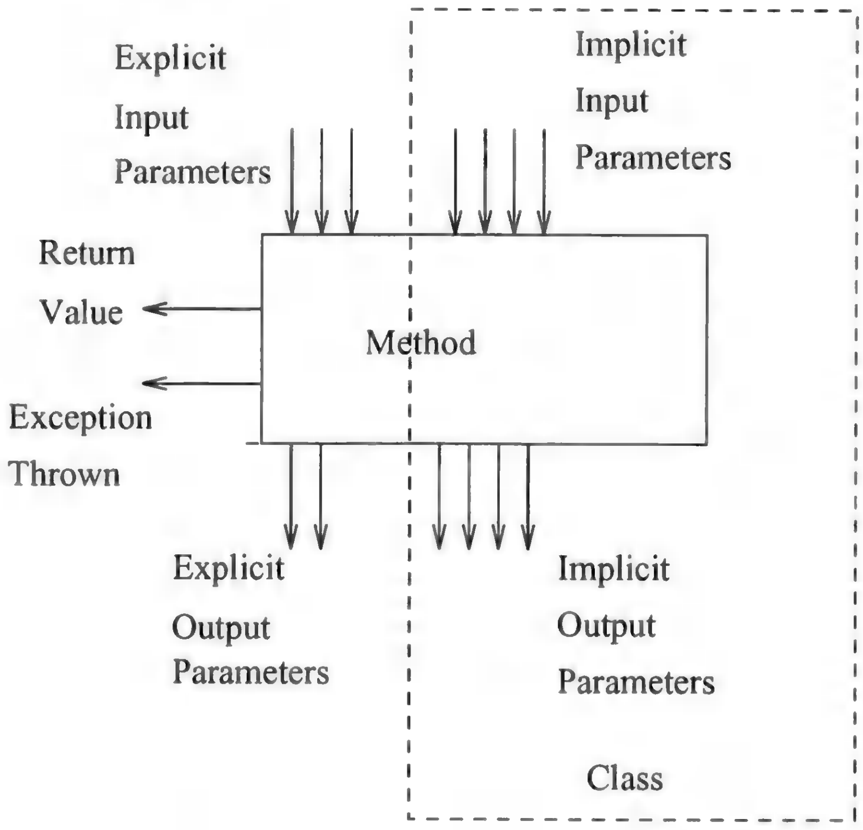


Figure 1: Method Inputs and Outputs

const objects in C++ or *final* objects in Java.

- **Constant method** - A constant method is one that does **not** modify any objects. This includes implicit and explicit parameters.

An explicit parameter can be either a **modifiable** or an **inspectable** object with respect to a specific method. The classification is determined by whether the object changes its state during the execution of the method. For the implementation of the metric being presented in this paper, all the implicit parameters are assumed to be **modifiable** objects. This metric is applied to the specification of a class which will not provide sufficient information to determine which of the implicit parameters might be modified. This assumption will make our metric value an upper bound on the effort estimation.

The attributes that are available for calculating the testability metric are the attributes defined in the class, the parameters declared in the method signatures and the attributes that are visible to the methods due to inheritance and aggregation.

3.3 Visibility Component of a Method

The Domain/Range Ratio metric proposed by Voas [24] is defined as the ratio of the cardinality of the possible inputs to the range of the possible outputs.

This metric is not directly applicable to object-oriented programs because the number of explicit parameters that are being passed to a method is not a real indication of the loss content in the method. This is because object-oriented languages discourage procedural autonomy and instead organize methods into a collection of operations that share the object state through non-local variables.

We therefore include the attributes defined in a class to be the **implicit parameters** to all the methods defined in the class. We take this conservative approch at the design phase due to lack of information about the actual set of objects that change their states at runtime. A reformulation of the metric for the coding phase would include only those objects that have been utilized. This redefinition would provide a greater accuracy of the metric.

Adapting the above metric, we introduce a concept termed the **visibility component (VC)** of a method. It defines the accessibility of objects that may have changed state during the execution of a method. We assume that the implicit and explicit parameters have equal potential to affect the object's current state.

The definition of the **visiblity component** ζ is defined as

$$\zeta = \frac{cardinality\_of\_possible\_outputs}{cardinality\_of\_possible\_inputs} \quad (1)$$

The following are considered to be *"Inputs"* in the calculation of the above metric.

- The number of explicit parameters in the method signature, if any.
- The number of implict parameters declared in the class.

The following are considered to be *"outputs"* in the calculation of the above metric.

- The explicit reference parameters in the method signature, if any.
- The implicit parameters, object attributes, defined in the class.
- The return value of the method, if any.
- Any exceptions thrown by the method.

Syntactically, C++ allows the definition in the class header of the exception objects that are thrown by a method. We therefore propose that the visibility component of that method includes the visiblity component of its exception objects too. The inherent assumption is that the exception objects that are thrown by the method include information about the state of the object and the correctness of execution just as do the implicit and explicit parameters to the method.

Future extensions of this work may include the calculation of information loss content based on the individual complexities of the thrown objects.

3.4 Testability of a Method

The **testability** of a method, η is dependent on the visiblity component of the method. We define the testability of a method in a class to be:

$$\eta = constant * (\zeta) \quad (2)$$

To make the testability estimate of a method a true indicator of the effort needed to test the method, we take the constant to be the number of unique objects that participate in the calculation of the metric. That is, we count the objects used as explicit input parameters, implicit input parameters, return objects and exception objects. We do not include the implicit output objects nor the pass by reference, explicit parameters in the count. This constant is a reasonable choice because its value indicates the maximum number of objects whose state may have been changed incorrectly.

3.5 Testability of a Class

We base our definition of the testability of a class on the testability values, η, of all the methods in the class. We define the testability of a class, θ, to be the **minimum** of the testabilities of the methods in the class.

$$\theta = min(\eta) \quad (3)$$

Since the testing effort is *maximum* in the method that hides the most information, we have selected the lower bound of the testabilities of all the methods, to be the testability of the class.
We do not include **constant objects and methods** in our complexity calculations. The reason is that the metric considers the set of ob-

jects that have the potential to change their state, possibly erroneously, during the execution of the method. Since constant objects and methods are **syntactically state invariant**, at least during the execution of the method, this assumption makes the metric more sensitive to those objects that may have taken on erroneous values that will be propagated on to other methods, and possibly other objects.

3.6 Inheritance

McGregor and Dyer[18] considered the inheritance relationship between two classes as a mapping that carries out certain transforms on attributes as they are inherited from one class to another. Each object-oriented language has its own set of transforms that may be applied.

This means that the attributes that are visible to the methods of the derived class are a subset of the attributes from the base class. The visibility component calculation therefore has to be modified to handle this.

In the case of C++, all of the **protected** and **public** attributes of the base class are visible to the derived class. This increases the set of data objects that are visible to the methods in the derived class. Other languages have different visibility and scoping rules, like the implicit private area in Smalltalk objects, and encapsulation mechanisms such as packages in Java. We therefore consider the set of implicit parameters in the derived class to include all the objects from the base class that are visible in the derived class.

$$implicit\_parameters = base\_class\_objects \bigcup derived\_class\_objects$$

These modifications also make the metric sensitive to the level of inheritance in the class hierarchy. Since the implicit parameter calculation is influenced only by the visibility level (i.e private, public or protected) of the objects in the base class rather than the class of the object, the metric is monotonically increasing. To address this issue, we have assumed that the count of the implicit parameters of a class at any level in the inheritance graph **includes** the protected/public attributes of all the classes that lie above it in the inheritance tree. If a class multiply inherits from many classes, the count of the implicit parameters that inherit from this class, increases substantially. This demonstrates that misuse of multiple inheritance could lead to potential problems in maintainance and testing.

The above counting rules also make the metric compatible with the DIT metric proposed by Chidamber[5]. The deeper a class in the hierarchy, the greater is the number of data objects it is likely to inherit, making it more complex to test.

3.7 Summary of Counting Rules

In this section we will summarize the set of counting rules that have been used for the calculation of the visibility component of a class.

- For a simple class without any inheritance, the set of implicit parameters is the set of all the private, public and protected data attributes in the class. The set of possible objects that can change state is limited to the set that is visible to the methods.

- For a class that inherits from another class, the set of implicit parameters is the set of all the data attributes that are defined in the class plus the set of implicit parameters from the base class except for any attributes that are private to the base class. The same counting rules are used while calculating the set of implicit parameters for a class that multiply inherits from different classes.

- Constant objects and methods are not considered in the complexity calculations. The metric is designed to estimate the set of objects that have the potential to change states.

- The Visibility Component of a method does not change as it is inherited into subclasses. Even though new attributes may be introduced, the method does not contain code to access these attributes and cannot modify them. Hence, the method treats these new attributes as constant objects.

- The Visibility Component will change monotonically as the method is redefined in subclasses. The signature of the method cannot be changed. Thus the only change to the computation of the metric is in the number of attributes that are implicit to the method. Since the metric calculationis solely based on the signature, changes in the implementation of the method do not influence the value. The output/input ratio will not change, but the testability will increase as the constant multiplier changes.

4 Example Calculations

The testability metric proposed in this paper can be used to estimate the effort that is required in the testing phase of the developmental cycle. Typically the parameters that are considered during effort estimation, include data due to metrication , maturity of the software developmental process within the company and the actual resources that are available for the project.

We propose that the effort that is needed to test a class hierarchy is the sum of all the resources expended in the testing phase of the project life cycle. One of the important factors that determine the effectiveness of testing cycle is the **coverage** of the testcases i.e the reach of the testcases in determining/verifying the set of objects that have changed states during execution. It is of statistical significance that the testing coverage is proportional to the number of test cases defined in the test plan. It is often the case that an increased activity of state changes within a method often correlates to an increase in the number of testcases to validate that method.

The proposed metric estimates, although conservatively, the number of objects that have the ability to change their states in a method. It is therefore logical to investigate the correlation between static and dynamic measures by measuring the actual objects that have changed states in a method.

5 Validation

5.1 Theoretical Validation

In this section we will continue to validate the measure theoretically, first by considering the response of the measure to the features specific to object-oriented systems. Then, in the rest of the section, we will consider a set of axiomatic properties which should be satisfied by any complexity measure. The complexity metric is then validated against these properties.

Object-Oriented Features Object-oriented designs have certain features that have been found to reduce the complexity of the software. Below we present these features and explain the response of the measure in those situations.

1. *Inheritance*: Whenever a class has to be developed we may have a choice of inheriting from an existing class or creating one from scratch. The measure increase the complexity of the class indicating that testing is always involved proposition when there is a lot of reuse.

2. *Information hiding*: By encapsulating its attributes or properties, a class restricts access to its information. This is often leads to loss of information which results in a greater complexity measure for the class. As the measure is inversly proportional to the loss component, it is very sensitive to feature of object-orientation. It is also necessary to note that the measure of the metric is dependent on the design of the class. If the methods in the class have a high visibility component then the measure decreases.

3. *Weak Coupling*: Classes should be as independent of each other as the domain they represent permits. Unnecessary interaction with other classes is considered bad design. Coupling is indicated by objects used as attributes within a class and as parameters to methods. The metric rewards weak coupling since it produces a lower value for classes that have fewer input and output parameters.

4. *Composition*: When the internal mechanism of a class uses the services of an already existing class through instantiation, the complexity of the containing class is reduced. This is the second form of reuse typical in object-oriented programs (the

first being reuse through inheritance.) Composition is an example of coupling which has already been addressed above.

Axiomatic Properties Weyuker[25] listed a set of properties (necessary but not sufficient) that are desired of a complexity measure for procedural programs. Although these properties have been criticized[1] , Cherniavsky [4], they do provide a basis for some validation of complexity metrics short of using experimental data. As shown by Weyuker, with such properties it is possible to filter out measurements with undesirable properties.

We have not used all of the properties defined by Weyuker. These properties were defined for metrics intended to measure procedural programs and, as such, define some constraints that are not applicable to object-oriented software. In particular, assuming a single entry, single exit module does not address the object-oriented aggregation of methods into a class. Where there is an obvious mismatch between our goals and the properties we have omitted the property.

In the list of properties below, E, F and G represent classes and the complexity of E computed by an ideal measure is represented by $\mid E \mid$. Classes can be concatenated either by *inheritance* or *composition* (collaboration). When a class F inherits from another class E, we depict the resulting subclass as $F \uparrow E$ Also, when a class E collaborates with another class F, the collaborating class is shown by $E \rightarrow F$. Two classes are said to be *equivalent* if they have the same behaviour for the classes.

- Property 1: $(\exists E)$ $(\exists F)$ $(\mid E \mid \neq \mid F \mid)$.
There exists two classes E and F such that the complexity of E is not equal to the complexity of F. The property requires that a measure should not produce the same value for every class. With our measure we can always come up with two classes with two different complexity values. We can always construct a class F which has a different number of data attributes or methods with different signatures from those in class E, such that the Visibility Component is different.

- Property 2: *Let c be a nonnegative number. Then there are only finitely many classes of complexity c.*
A measure has to be sufficiently sensitive, and this property requires that a measure have a bounded range of values that it can assign. By the principle of number theory we can show that there is a finite combination of series whose sum is equal to $(\mid c \mid)$. Each series in the combination represents a particular set of attribute Visibilities. Since there is a finite number of distinct series, the number of classes with the Visibility Component, c is finite.

- Property 3: *There are distinct classes E and F such that,* $\mid E \mid = \mid F \mid$.
A measure that assigns a unique value to every class is not much of a measure. It would go against the principle of measurements which requires that the number of objects that can be measured be greater than range of the values of the measure. Our measure clearly satisfies this property, since the Visibility Component does not depend on the types of the attributes. Given a class E, we can create a new class with a different number of attributes and methods having high visiblities so that the new class F has the same complexity value.

- Property 4: $(\exists E)$ $(\exists F)$ $(E \equiv F$ *and* $\mid E \mid \neq \mid F \mid)$.
There exist classes E and F such that E is equivalent to F but the complexity of E is not equal to the complexity of F. There are different implementations for the same functionality. Since our metric does not depend on data types, it independent on class implementations. Therefore this property is satisfied by our metric.

- Property 5: $(\forall E)$ $(\forall F)$ $(\mid F \mid \leq \mid F \uparrow E \mid)$.
For all classes E and F, if F inherits from E then the complexity of F ↑ E is greater than or equal to the original complexity of F. Due to the inheritance, class F has access to all the public and protected data

[1]The major drawback of Weyuker's properties is that definition of complexity is given.

attributes of E. This increase in the implicit parameters of class F making the measure sensitive to inheritance.

Since our measure happens to satisfy all the above properties it can be considered to have passed a significant part of the theoretically validation process. The measure has been tested againt those axioms proposed by Weyuker that are applicable to both procedural and object-oriented software systems.

5.2 Experimental Validation

The most conclusive validation of the metric presented in this paper would be to collect data from actual development projects. Since the VC metric presented in this paper is intended to give an estimate of the effort required to test a class, a possible experiment would be to apply the metric to a set of classes, predict the amount of effort required to test each one and then to have each class tested by a developer. The actual effort required to test each could be obtained and the accuracy of the prediction could be measured.

Since such experiments are difficult to control and even more difficult to have approved, we propose an alternative approach in which the number of test cases required to achieve a specified level of test coverage is used to estimate the total testing effort. For our experiment we used **branch coverage**[2] as our coverage criteria. The criteria chosen is irrelevant provided that it is recognized as a valid, and likely, technique that might be used by developers on an actual project.

One criteria for the level of test coverage that is important is that the testing technique should utilize the same level of information as VC. The value of VC presented here is the *internal* visibility because it considers attributes that are not visible external to the object. This estimate would approximate the amount of effort that would be devoted to testing the individual methods. This internal view corresponds more closely to a structural testing approach, such as branch testing, than to a specification-based approach. In the conclusion section we will consider the *external* visibility of an object that corresponds to specification-based testing.

For our experiment, we chose classes from several sources including locally written code and widely used libraries. The VC was calculated for each method and each class. Each modifier method was examined and the number of branch test cases needed to achieve 100% branch coverage of each method was calculated.

A non-parametric correlation statistic, Spearman's rho, was used to determine the amount of agreement between the number of test cases required to give 100% branch coverage and the internal visibility. This comparison resulted in a correlation of .35. The significance of the correlation was investigated using the student's t distribution due to the sample size. The correlation was found to be significant at the .1 confidence level.

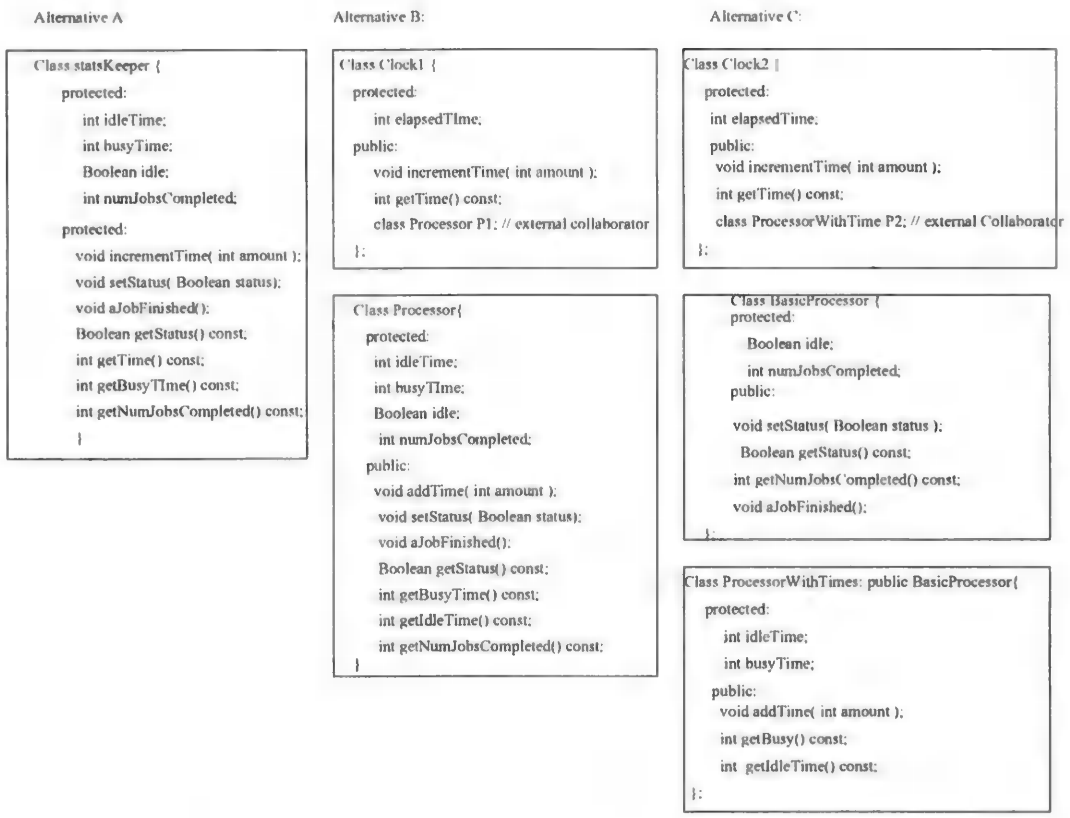
Initial comparisons were less successful than the present one because they included both accessor and modifier methods. Accessor methods confound the VC metric in that there is more information flowing out than flowing in. Accessor methods are also very simple structurally and usually have only one path; therefore, the testing effort is negligible. For this experiment and those reported below, accessor methods were excluded from the calculation.

Several other experiments have been conducted to experimentally validate the metric.

The testability of a method is directly related to the complexity of its structure[14]. Using the original set of classes, we computed the cyclomatic complexity for each method. Once again there was statistically significant agreement between VC and the cyclomatic complexity.

In a previous study[1], a set of design problems was presented to a group of developers experienced in object-oriented design. Figure 2 presents one set of design solutions for a problem in tracking the idle time for a set of processors. The developers were asked to rank three solutions to each design problem. These rankings were then compared to the rankings computed using a complexity metric, Permitted Interactions. In a subsequent study another metric, intended for use even earlier in the development process, was compared to these expert rankings. In each case, statistically significant agreement was found.

For the testability metric, a similar compar-



Alternative A

```
Class statsKeeper {
    protected:
        int idleTime;
        int busyTime;
        Boolean idle;
        int numJobsCompleted;
    protected:
      void incrementTime( int amount );
      void setStatus( Boolean status);
      void aJobFinished();
      Boolean getStatus() const;
      int getTime() const;
      int getBusyTIme() const;
      int getNumJobsCompleted() const;
      }
```

Alternative B:

```
Class Clock1 {
  protected:
     int elapsedTIme;
  public:
     void incrementTime( int amount );
     int getTime() const;
     class Processor P1; // external collaborator
 };
```

```
 Class Processor{
   protected:
    int idleTime;
    int busyTIme;
    Boolean idle;
     int numJobsCompleted;
   public:
    void addTime( int amount );
     void setStatus( Boolean status);
     void aJobFinished();
     Boolean getStatus() const;
     int getBusyTime() const;
     int getIdleTime() const;
     int getNumJobsCompleted() const;
 }
```

Alternative C:

```
Class Clock2 {
   protected:
    int elapsedTime;
    public:
     void incrementTime( int amount );
     int getTime() const;
     class ProcessorWithTime P2; // external Collaborator
  };
```

```
      Class BasicProcessor {
      protected:
         Boolean idle;
         int numJobsCompleted;
      public:

       void setStatus( Boolean status );
         Boolean getStatus() const;
       int getNumJobsCompleted() const;
       void aJobFinished();
  };
```

```
Class ProcessorWithTimes: public BasicProcessor{
   protected:
      int idleTime;
      int busyTime;
    public:
     void addTime( int amount );
     int getBusy() const;
     int  getIdleTime() const;
 };
```

Figure 2: Example Solutions to a Design Problem

ison was made between the rankings by experts and the VC metric. Figure 3 shows the visibility calculations for the design alternatives. The results of the comparison are shown in table ??. The VC metric agreed with the expert rankings in 14 of the 17 cases. In two of the three cases for which there was disagreement, previous studies[1, 12] had also shown some disagreement indicating that these examples may have alternatives that are too closely related.

5.2.1 Analysis

The relationship between VC and the number of branch test cases is weak; however, the combination of validations that we used, taken together, provide compelling evidence for this approach.

One reason for the weak relationship may be due to the lack of sensitivity of VC. A large number of the methods that were examined had ratios of 1. We partially addressed the lack of sensitivity issue by multiplying the ratio by the number of unique objects associated with a method. We will further discuss the sensitivity issue in the future work section of the paper. The lack of sensitivity is not surprising given the stage in the development process at which we wish to apply the metric. Early models, of necessity, contain less exact information than later models, such as code; however, estimates of project size and effort must be made as early as possible with the best information available.

A post hoc analysis revealed an interesting relationship. The agreement between measures for the base class and the internal visibility is very high (for one set of classes the correlation was 1.0). This agreement declines as classes

Alternative A

Class StatsKeeper

Name of the method	Visiblity Component
incre.entTime()	5/4 = 1.25
setStatus()	5/4=1.25
aJobFinished()	4/4=1

Testabality of Alternative A is: 1.25

Alternative B:

Class Processor:

Name of the method	Visiblity Component
addTime()	5/4 = 1.25
setStatus()	5/4 = 1.25
aJobFinished()	4/4 = 1.0

Class Clock1

Name of the method	Visibility Component
incrementTime	2/1 = 1

Testability of Alternative B is: 2.00

Alternative C

Class BasicProcessor

Name of the method	Visibility Component
setStatus()	3/2 = 1.5
aJobFinished()	2/2 = 1.0

Class ProcessorWithTimes

Name of the method	Visiblity Component
addTime()	5/4 = 1.25

Class Clock2

Name of the method	Visibility Component
incrementTime()	2/1 = 2

Testabality of Alternative C is: 2.00

Figure 3: Example Calculations for VC

lower in the hierarchy are added to the computation.

6 Conclusion

The proposed testability metric estimates the likelihood that an object will reveal its faults. The metric is sensitive to changes in the number of data attributes and the depth of the inheritance. By making the metric dependent on the information content of the class, as represented by the signatures of the methods, rather than the structure of the class, the metric may be used in the early phases of the development cycle.

Although the sensitivity of the metric is less than it would be if applied later in development when the code is available, it is our opinion that the metric provides a sufficient basis for estimating the resources and effort necessary for adequate testing of the code.

There are several possible extensions of this work. These include:

- **Sensitivity studies** - The metric as currently defined is not very sensitive. A large number of methods cluster around metric values of 0, 1 and infinity[2]. Possible redefinitions could include weights for the individual attributes of a class, such as its methods and attributes to reflect their participation in the state of the object.
- **External visibility** - The external visibility is analogous to the internal visibility for a method/class. The external visibility does not include the implicit attributes of the class. The relationship between this measure and a specification-based testing technique is being investigated.

[2]Infinity is the result when a method has no inputs but some outputs.

Comparison	Expert Pref.	PC Pref.	PI Pref.	VC Pref.	
RV2 vs RV3	RV3	RV3	RV3	RV3	√
RV1 vs RV3	RV3	RV3	RV3	RV3	√
ST1 vs ST2	ST2	ST2	ST2	ST2	√
ST2 vs ST3	ST3	ST3	ST3	ST3	√
ST1 vs ST3	ST3	ST3	ST3	ST3	√
R1 vs R2	R2	R1 or R2	R2	R1 or R2	√
S1 vs S2	S2	S2	S2	S1	×
Q1 vs Q2	Q2	Q1	Q2	Q1	×
Q2 vs Q3	Q2	Q2	Q2	Q2	√
Q1 vs Q3	Q1	Q1	Q1	Q1	√
Q4 vs Q5	Q4	Q5	Q4	Q4	√
Q5 vs Q2	Q2	Q5	Q5	Q2	√
Q4 vs Q2	Q4	Q2 or Q4	Q4	Q2	×
IO1 vs IO2	IO2	IO1	IO2	IO2	√
IO2 vs IO3	IO2	IO2	IO2	IO2	√
IO1 vs IO3	IO3	IO1	IO1	IO3	√
TT1 vs TT2	TT2	TT1	TT1	TT1 or TT2	√

Table 1: Preferences by experts, PC,PI and VC

- **Variability of the measure** - The difference between accessor and modifier methods was obvious and treating them differently was easily justified. Other factors that affect the testability may be worthy of study. For example, in our data analysis, we noted informally that the visibility measure decreased as the depth of the class in the inheritance hierarchy increased. These observations may lead to better understanding of other less obvious relationships.

Although this metric has not been tested on a full scale software project, our preliminary results from applying the metric on a wide range of C++ classes have been successful both theoretically and experimentally. It has been defined in such a way that it will fit into our framework for iterative metric management[17]. In this framework an early measure, such as VC, is paired with measures that can only be applied later in the life cycle, such as Voas's information loss metric[24] or the simpler McCabe's Cyclomatic Complexity[14] metric. With this framework and automated metric collection tools, a manager can estimate early and measure exactly later in the life cycle, to provide a continuous picture of the project.

References

[1] D. H. Abbott, T. D. Korson, and J. D. McGregor. A proposed design complexity metric for object-oriented development. Technical Report TR 94-105, Clemson University, 1994.

[2] Boris Beizer. *Software Testing Techniques*. International Thompson Computer Press, second edition, 1990.

[3] Antonio Bertolino and Lorenzo Strigini. On the use of testability measures for dependability assessment. *IEEE Transactions on Software Engineering*, 22:97–108, 1996.

[4] J. C. Cherniavsky and C. H. Smith. On weyuker's axioms for software complexity measures. *IEEE Trans. on Software Eng.*, 17(6):636–638, June 1991.

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design.

IEEE Trans. on Software Eng., 20(6):476–493, June 1994.

[6] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. on Software Eng.*, 20(3):199–206, March 1994.

[7] N. E. Fenton. When a software measure is not a measure. *IEE Software Eng. J.*, 7(5):357–362, May 1992.

[8] Renato R Gonzalez. A unified metric of software complexity: Measuring productivity and value. *Journal of Systems Software.*, 29:17–37, 1995.

[9] Warren Harrison. An entropy-based measure of software complexity. *IEEE Trans. on Software Eng.*, 18(11):1025–1029, November 1992.

[10] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Trans. on Software Eng.*, 7(5):510–518, September 1981.

[11] Vernon. V. Chatman III. Change points: A proposal for software productivity measurement. *Journal of Systems Software.*, 31:71–91, September 1995.

[12] Sanjay Kamath and John D. McGregor. A measure of psychological complexity. Technical report, Dept. of Computer Science, Clemson University, 1995.

[13] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics.* Prentice-Hall, 1994.

[14] T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, December 1976.

[15] J. D. McGregor. Managing metrics in an iterative incremental development environment. Technical Report TR 94-110, Clemson University, 1994.

[16] J. D. McGregor and D. A. Sykes. *Object-oriented Software Development: Engineering Software for Reuse.* Van Nostrand-Reinhold, New York, NY, 1992.

[17] John D. McGregor. Managing metrics in an iterative environment. *Object Magazine*, 5(6):65 – 71, 1995.

[18] John. D. Mcgregor and Douglas M Dyer. A note on inheritance and state machine. *ACM SIGSOFT SOFTWARE ENGINEERING NOTES*, 18(4):61–69, October 1993.

[19] John D. McGregor and Anuradha Kare. A parallel architecture for component testing. In *Proceedings of the Ninth International Quality Week*, 1996.

[20] Neeraj Ojha and John D. McGregor. Object-oriented metrics for early system characterization: A crc card-based approach. Technical Report TR 94-107, Dept. of Computer Science, Clemson University, 1994.

[21] M. Shepperd. An evaluation of software product metrics. *Info. and Software Technology*, 30(3):177–188, April 1988.

[22] M. Shepperd and D. Ince. *Derivation and Validation of Software Metrics.* Clarendon Press, Oxford, 1993.

[23] Jeffery Voas, Larry Morell, and Keith Miller. Predicting where faults can hide from testing. *IEEE Software.*, pages 41–48, March 1991.

[24] Jeffery M. Voas and Keith W. Miller. Semantic metrics for software testabality. *Journal of Systems Software*, 20:207–216, 1993.

[25] E. J. Weyuker. Evaluating software complexity measures. *IEEE Trans. on Software Eng.*, 14(9):1357–1365, September 1988.

Design Patterns for Dealing with Dual Inheritance Hierarchies in C++

Robert C. Martin
Object Mentor
rmartin@oma.com

INTRODUCTION

Dual hierarchies are a common element of significant Object-Oriented applications, They arise out of the need to separate concerns. Despite their prevalence, they present problems to the designer that are often solvable only by using techniques that are generally considered unsafe.

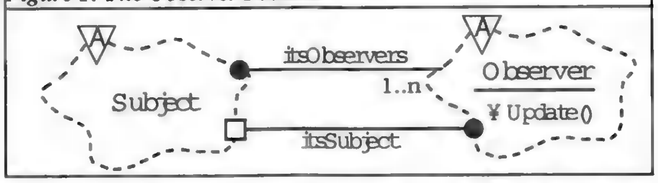
This paper presents three patterns that can be employed to deal with the problems of dual hierarchies. The patterns may be used in isolation, to solve specific application related problems, or they can be used together as a small pattern language in order to more comprehensively address the issues in a larger application.

The patterns are called: RUNGS OF A DUAL HIERARCHY, INTELLIGENT CHILDREN, and STAIRWAY TO HEAVEN. In addition, another pattern, RTTI VISITOR, is presented as an ancillary pattern that supports the others.

WHAT IS A DUAL HIERARCHY?

Consider the OBSERVER[1] pattern (Figure 1). This pattern is often employed when the actions precipitated by the change of an object's state must be disassociated from that object. The changed object derives from the `Subject` class which simply sends a `Notify` message to the abstract `Observer` interface. Derivatives of `Observer` implement the appropriate actions.

Figure 1: The Observer Pattern



Typically, the `Observer` derivative must communicate back to the changed object, so it holds a pointer or reference to it.

The code for these classes is shown in Listing 1.

Listing 1: Observer Classes

```
class Observer;

class Subject
{
  public:
    void Register(Observer& o);
    void Notify() const;

  private:
    Set<Observer*> itsObservers;
```

1. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994, p. 293

Listing 1: Observer Classes

```
};

void Subject::Register(Observer& o)
{itsObservers.Add(&o);}

void Subject::Notify() const
{
    for (Iterator<Observer*> i(itsObservers); i; i++)
        (*i)->Notify();
}

class Observer
{
  public:
    Observer(Subject& s) : itsSubject(s) {};
    Subject& GetSubject() const {return itsSubject;};
    virtual void Notify() = 0;
  private:
    mutable Subject& itsSubject;
};
```

Now consider how this pattern might be employed with a `Clock` object (Figure 2) to display the time of day on a screen. `Clock` is a simple concrete class whose task is to keep track of the time. `Clock` is a highly reusable object, so we don't want it to have to know about `Observer`. So we derived `ObservedClock` from both `Clock` and `Subject`.

The methods of `Clock` that change its state are overridden in `ObservedClock` so that they will delegate to `Clock`, and also inform the `Subject` of the state change. The `ClockObserver` is notified when the `ObservedClock` is changed. It then procures the current time from the `Clock` and displays it on a screen.

Figure 2: Clock Observer

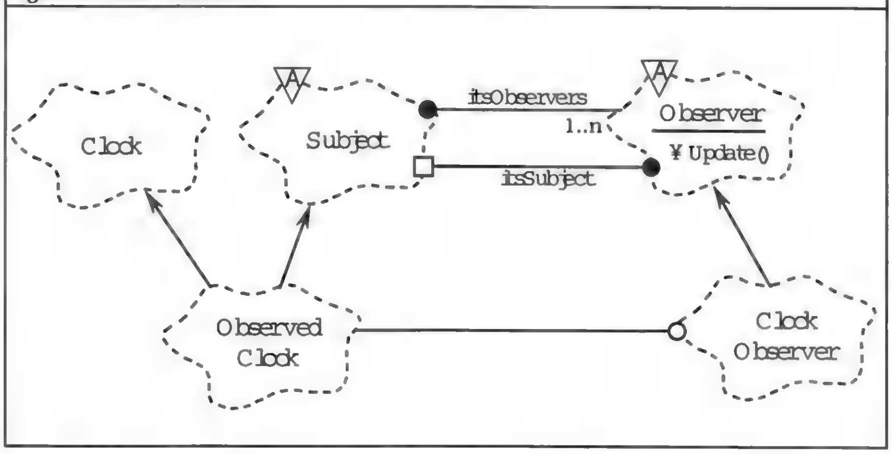


Figure 2 is a typical dual hierarchy. The `Subject` hierarchy and the `Observer` hierarchy are extremely similar. For every derivative of `Subject` there will be a corresponding derivative of `Observer`. Notice that the reason this dual hierarchy was created was to separate, from `Clock`, any concern about the type of actions, or number of actions that are precipitated by a change in the state of the `Clock` object. It is this separation of concerns that typically causes dual hierarchies to be created.

The problem of static typing.

Consider the code for `ClockObserver` in Listing 2. Note that the `Notify` function of this class must downcast the `Subject` object that it gets from its base class `Observer`. This is the traditional typing problem caused by dual hierarchies in languages that are statically typed.

Listing 2: ClockObserver

```
class ClockObserver : public Observer
{
  public:
    ClockObserver(ClockSubject& s);
    virtual void Notify();
};

ClockObserver::ClockObserver(ClockSubject& s)
: Observer(s)
{}

void ClockObserver::Notify()
{
    Clock& c = (Clock&) GetSubject(); // ugly downcast
    Display(c.GetTime());
}
```

The typing problem does not exist in dynamically typed languages since the language system does not perform any static checks upon the type. In a dynamically typed language the derived `Observer` would simply send the messages that it expects the derived `Subject` to be able to respond to.

In statically typed language the typing problem stems from the fact that corresponding elements of the two hierarchies know about each other, but their base classes do not know about the derivatives. Since `Observer` contains a reference to a `Subject`, derivatives of `Observer` must downcast the `Subject` to the corresponding kind.

The INTELLIGENT CHILDREN pattern addresses this issue.

INTELLIGENT CHILDREN

Intent

When using a statically typed language, this pattern provides a way to avoid downcasting in some dual inheritance hierarchies.

Motivation

In a dual hierarchy rooted at the base classes B1 and B2, if B1 contains a reference to B2, and if derivatives of B1 gain access to derivatives of B2 through this pointer, then those objects must be downcast before they can be used as their true type. It is best to avoid downcasting where possible.

Solution

Move the contains relationship from the bases to the derivatives. That is, have the derivative D1 contain a reference to D2. Demote the relationships between the bases to a using relationship. Provide a pure interface in the base classes that acts as an accessor function. The derived classes can implement that pure interface.

Structure

See Figure 3.

Applicability

Not all dual hierarchies can yield to this simple approach. Many must procure derived objects from sources that only the base peers have access to. In such cases, INTELLIGENT CHILDREN is not a viable option. However, when it is possible to create the derived peers such that they know about each other, or when the knowledge of a peer can be passed directly to the other peer without going through a base, then INTELLIGENT CHILDREN is a good option.

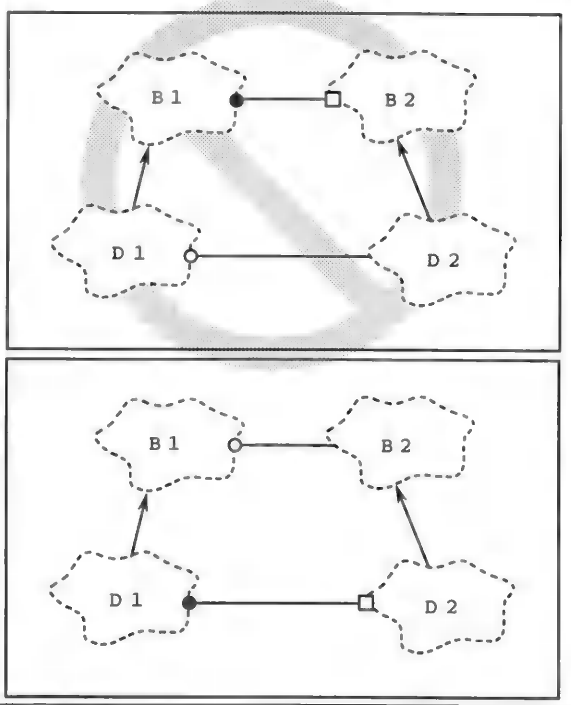
Sample Code

See Listing 3

Listing 3: Intelligent Children

```
class B1
{
  public:
    virtual B2& GetB2() const = 0;
};

class B2 {};
class D2 : public B2 {};

class D1 : public B1
{
  public:
    D1(D2& d2) : itsD2(d2) {};
    virtual D2& GetB2() const {return itsD2;}

  private:
    mutable D2& itsD2;
};
```

Figure 3: Intelligent Children



Consequences

The most serious consequence of this pattern is a minor warping of the object model. Consider Figure 3 again. Note that the upper diagram is a better representation of the abstract concept. All B1 objects contain a reference to a B2 object. The presence of the containment at this level makes it impossible for a derivative of B1 *not* to contain a derivative of B2. So by moving the containment to the derived peers we are creating an opening for a B1 derivative that does not contain a B2 derivative.

This is mitigated to some extent by the presence of the pure accessor interface. If B1 has a pure `GetB2()` interface, then it is likely that derivatives of B1 will have some way to procure a B2 instance, even if it is not through containment.

There is a burden placed upon the programmer to remember to include this containment in every derivative of B1.

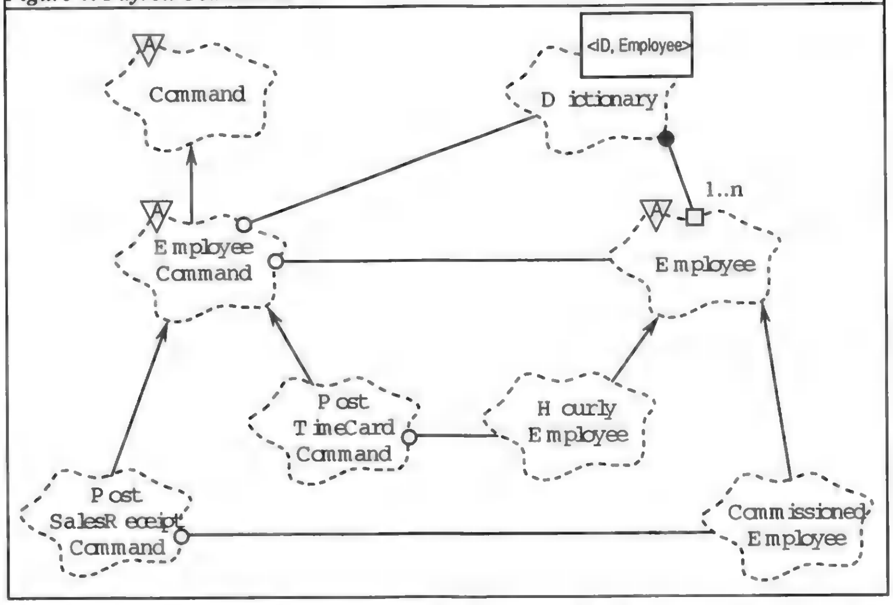
DYNAMIC DUAL HIERARCHIES

A dual hierarchy is like a ladder. The two inheritance hierarchies are the supports of the ladder, and in the INTELLIGENT CHILDREN pattern above, the contains relationships in the peers are the rungs of the ladder. However there are dual hierarchies in which the rungs cannot be supported with simple contains relationships. In such hierarchies the peers do not retain their association indefinitely. Instead, peer objects are associated for a period of time and then the association is broken.

If the association of the two peers takes place at the base level, then there is no way that INTELLIGENT CHILDREN can be employed to prevent the required downcast. As an example, consider the COMMAND[1] pattern. A `Command` object is generally transient. It is usually associated with other objects that it operates upon. And it is often found in a hierarchy that parallels the objects that it operates on.

I will draw upon the Payroll example from my recent book[1]. See Figure 4. Here we see that there is a hierarchy of `Command` classes. We also see that there is a hierarchy of `Employee` classes. The dual nature of the hierarchy is clearly evident. `EmployeeCommand` is associated with `Employee`, `PostTimeCardCommand` it associated with `HourlyEmployee` and `PostSalesReceiptCommand` is associated with `CommissionedEmployee`.

Figure 4: Payroll Commands



Note that all the `Employee` objects are contained in a `Dictionary` that associates the objects with an ID. This ID is used by the `EmployeeCommand` object to locate the appropriate `Employee` object to operate upon. Thus the `EmployeeCommand` class is invoking the TEMPLATE METHOD[1] pattern by factoring the access of `Employee` objects out of the derived commands. Having fetched the `Employee` object the `EmployeeCommand` class calls its own pure interface `ProcessEmployee` which it expects the derivatives to implement.

Here we face the same downcasting problem that we faced with the OBSERVER pattern, when a `PostTimeCard-Command` object gets a `ProcessEmployee` message it is passed an `Employee` object which it must downcast to a `HourlyEmployee` object. However, this time we cannot employ INTELLIGENT CHILDREN to avoid the downcast because the peers are not permanently associated, and the temporary associations are being built by the base class `Em-ployeeCommand` which does not know about the derivatives.

1. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 233
1. *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prentice Hall, 1995.
1. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 325

Although this is primarily a problem associated with statically typed languages, this situation cannot be ignored by programmers of dynamically typed languages. The risk is that a `PostTimeCardCommand` will one day be asked to operate on the ID of a `CommissionedEmployee` object.

The RUNGS OF A DUAL HIERARCHY pattern addresses this situation by specifically permitting a type checked downcast.

RUNGS OF A DUAL HIERARCHY

Intent

To provide a context that justifies the use of type checked dynamic downcasting in dual hierarchy situations.

Motivation

Downcasting, even type checked downcasting, is often considered a poor engineering practice[1]. However, there are situations in which there is no alternative to using a downcast. This pattern is motivated by the need to identify one of those situations.

Solution

When programming a dual hierarchy in which peer to peer associations are built dynamically by objects that are not aware of the derived peers (i.e. objects at a level of abstraction that is higher than the peers) then type checked downcasting is an acceptable mechanisms for determining if the association has been built correctly. Moreover, in statically typed languages the type checked downcast is the only way to safely gain access to the specific methods of the peers.

Sample Code

```
class B1
{
  public:
    B2* GetB2() const; // cannot be overridden
};

class B2 {};
class D2 : public B2
{
  public:
    void D2Operation();
};

class D1 : public B1
{
  public:
    void DoOperation();
};

void D1::DoOperation()
{
    B2* b2 = GetB2();
    D2* d2 = dynamic_cast<D2*>(b2);
    if (d2)
    {
        d2->D2Operation();
    }
}
```

1. The reason for this is beyond the scope of this article, but has to do with potential violations of the open/closed principle of object oriented design. See: *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prehtice Hall, 1995, p. 286 and p. 360

Notes

There is much debate over whether type checked downcasts are appropriate tools for use in object oriented software. This controversy stems from the fact that there are many uses to which type checked downcasts *could* be put that are better addressed with polymorphic message dispatch; as in the INTELLIGENT CHILDREN pattern. However, there are cases, such as those described in this pattern, where polymorphic dispatch can not resolve the issue. In those cases, type checked downcasts are appropriate.

The type checked downcast can be implemented in many different ways. Many languages have some form of query to check if an object conforms to a certain type, or responds to a certain interface. In languages that do not support this directly, or in which the type check operation is too expensive for the algorithm that uses it, unique type values can be added to all the derived objects.

Another option is to use the RTTI VISITOR pattern described below.

Applicability

Use this pattern whenever you are programming a dual hierarchy and there is no way to retain the type identity of the derived peers in the peers themselves.

RTTI VISITOR

Intent

This pattern provides a mechanism by which type checked downcasting can be implemented in languages that do not directly support it. The technique mapped out here is also very fast.

Motivation

Although type checked downcasting is often considered inappropriate, and sometimes even dangerous, there are still some situations that require its use. Not all languages provide a mechanisms for type checked downcasting.

Solution

Use the VISITOR[1] pattern to provide classes that can be invoked by global downcast functions.

Structure

See Figure 5.

Sample Code

See Listing 4

Notes

The dependency structure of the VISITOR pattern is cyclic. That is, the base class depends upon the visitor class. The visitor class depends upon the derived classes. The derived classes depend upon the base class through the inheritance relationship, thus completing the cycle.

What this means is that users of any of the derived classes must be recompiled (and possibly retested) whenever *any* other derivative changes. Thus, this pattern must be used with care.

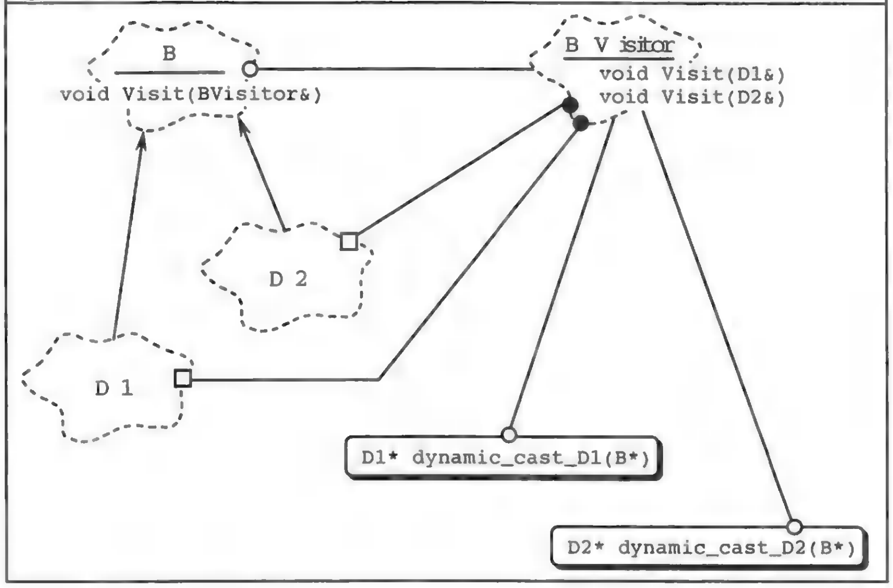
Applicability

This pattern is most applicable in situations where type checked downcasting is required and either:

1. The implementation language does not directly support type checked downcasting
2. The type checked downcasting supported by the language is too expensive to use where needed.

1. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 331

Figure 5: RTTI Visitor



Listing 4: RTTI Visitor

```
class BVisitor;
class D1;
class D2;

class B
{
  public:
    virtual void Visit(BVisitor&) = 0;
};

class BVisitor
{
  public:
    BVisitor()
    : itsD1(0)
    , itsD2(0)
    {}

    void Visit(D1& d1) {itsD1 = &d1;}
    void Visit(D2& d2) {itsD2 = &d2;}
    D1* GetD1() const {return itsD1;}
    D2* GetD2() const {return itsD2;}

  private:
    D1* itsD1;
    D2* itsD2;
};

class D1 : public B
{
  public:
    virtual void Visit(BVisitor& bv)
    {bv.Visit(*this);} // class Visit(D1&)
};

class D2 : public B
```

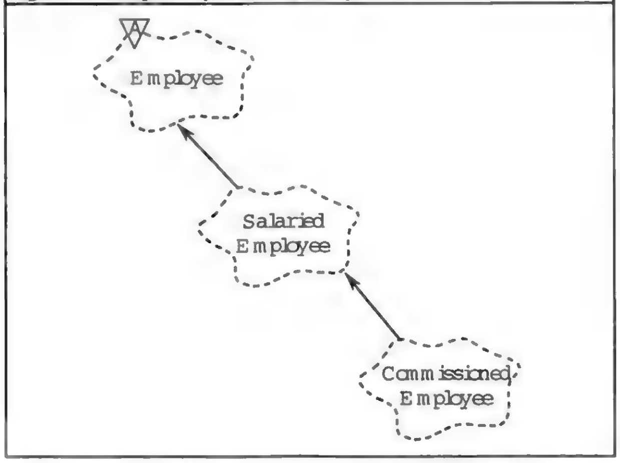
Listing 4: RTTI Visitor (Continued)

```
{
  public:
    virtual void Visit(BVisitor& bv)
    {bv.Visit(*this);} // class Visit(D2&)
};

// Global functions
D1* dynamic_cast_D1(B* b)
{
    BVisitor v;
    b->Visit(v);
    return v.GetD1();
}

D2* dynamic_cast_D2(B* b)
{
    BVisitor v;
    b->Visit(v);
    return v.GetD2();
}
```

DUAL HIERARCHIES OF INTERFACE AND IMPLEMENTATION

Consider a use of the ADAPTER[1] pattern. We have a hierarchy of classes that model part of the payroll problem (See Figure 6) and we would like to ADAPT these classes to a class named `PersistentObject` which provides the methods and facilities for writing objects out onto some persistent storage device.

Figure 6: Simple Payroll Hierarchy



The `Employee` class and the `PersistentObject` class can be ADAPTED by inheriting both of them into a `PersistentEmployee` class. It is clear that `PersistentSalariedEmployee` should inherit from `SalariedEmployee`, but it must also inherit from `PersistentEmployee`. Likewise, `PersistentCommissionedEmployee` must inherit from `CommissionedEmployee` as well as `PersistentSalariedEmployee`. This, is the STAIRWAY TO HEAVEN pattern (See Figure 7)

1. *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 139

Stairway to Heaven

Intent

This pattern describes the network of inheritance relationships that is needed when a given hierarchy must be adapted, in its entirety, to another class.

Motivation

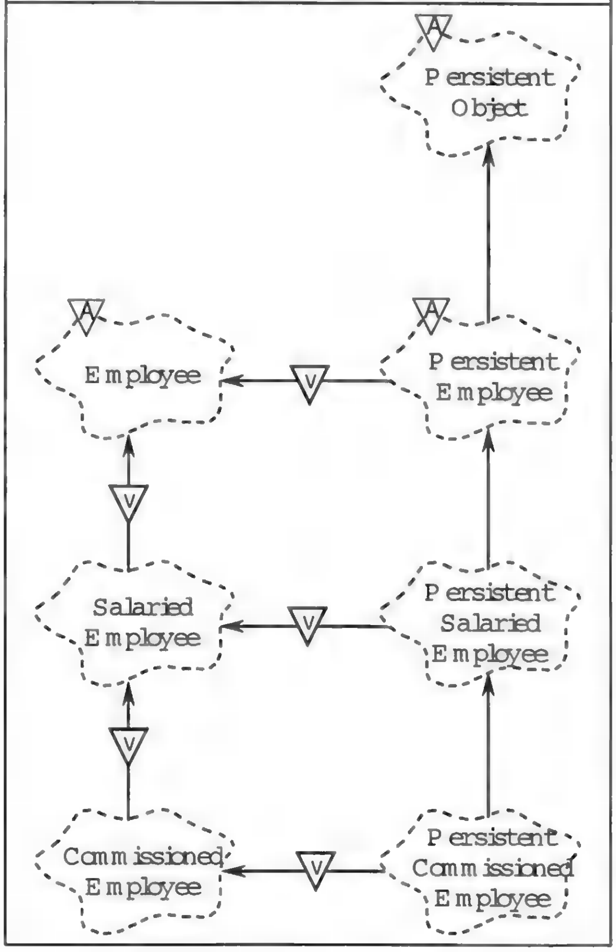
If a class hierarchy is to be reusable, it cannot depend upon detailed implementations. For example, one might have a class hierarchy of payroll objects such as: Employee, SalariedEmployee, HourlyEmployee, etc. If this these classes focus only upon the algorithms necessary to implement their particular abstractions, then they are highly reusable. However, if they were to incorporate the methods for reading and writing such objects to a particular database engine, then they would not be reusable in applications that did not have access to, or need that particular engine. Thus, we would like to keep any knowledge of database engines out of these objects.

To do this, a new set of classes needs to be created that inherits the ability to read and write themselves using the particular database engine required by the application, and the methods that model the payroll abstraction. This keeps the payroll objects separate and reusable.

Structure

See Figure 7

Figure 7: Payroll Stairway to Heaven



Notes

Figure 7 shows the use of virtual inheritance within the structure of this pattern. This is a C++ -ism, but has correspondents in other languages. The use of virtual inheritance is necessary in this pattern to prevent the repeated inheritance of the base classes. The desire is that there be only one copy of all the base objects in any of the derived objects.

Applicability

This pattern is best used to insure the isolation of concepts so that reusable class hierarchies do not become polluted with concepts that are application specific.

CONCLUSION

This paper has discussed three patterns that can be used to manage dual hierarchies. We have noted that dual hierarchies arise because of the desire to separate orthogonal concepts from each other. Most typically this allows reusable classes to remain independent of application specific details.

One thing that I found especially interesting while writing this paper, was the number of other patterns that I used in my examples. When determining the best way to present the problems of dual hierarchies to a group of pattern literate readers, the answer in each case was to relate the problems to a different pattern. This shows that patterns are effective ways for engineers to reason about software problems. It also demonstrates that there are deeper relationships between patterns than we can see by examining them in isolation.

The Object Group Design Pattern

Silvano Maffeis*
maffeis@acm.org

Olsen & Associates
Zurich, Switzerland

Abstract

This paper describes "Object Group", an object behavioral pattern for group communication and fault-tolerance in distributed systems. The Object Group pattern supports the implementation of replicated objects, of load sharing, and of efficient multicast communication over protocols such as IP-multicast or UDP-broadcast. Application areas of the pattern are fault-tolerant client/server systems, groupware and parallel text retrieval engines. Events within an Object Group honor the Virtual Synchrony model. With Virtual Synchrony, the size of an object group can be varied at run-time while client applications are interacting with the group. A replicated state remains consistent in spite of objects entering and leaving the group dynamically and in spite of failures. The Object Group pattern has been implemented in the Electra and in the Orbix+Isis CORBA Object Request Broker.

Keywords: Distributed Systems, Design Patterns, Replication, Group Communication, Virtual Synchrony.

1 Intent

The Object Group design pattern provides a local surrogate for a *group* of objects distributed across networked machines. The members of an object group have a consistent view of which other objects are also part of the group and are notified of objects joining and leaving the group. The pattern also provides a high-level, object oriented interface to low-level multicast facilities such as Ethernet, ATM, UDP-broadcast and IP-multicast.

*Supported by the Swiss National Science Foundation (Schweizer Nationalfonds) while on leave at Cornell University, Dept. of Computer Science.

2 Also Known As

Server Group Service, Replicated Object Pattern, Active Replica Pattern.

3 Motivation

To illustrate the object group pattern, consider the following CORBA IDL interface:

```
// IDL
interface Directory {
    void install(in string key, in any value)
        raises (ENTRY_EXISTS);
    void lookup(in string key, out any value)
        raises (NO_SUCH_ENTRY);
    void remove(in string key, out any value)
        raises (NO_SUCH_ENTRY);
};
```

The `Directory` interface could be used to implement an online phone book service. To ensure high availability of the service, several copies of the `Directory` are created on different hosts and are joined to an object group. Client applications use one object reference to represent the whole group without being concerned about the references of the individual group members.

The `remove` and `install` operations are multicast to the group because these modify the replicated state of the directory. A `lookup` operation, on the other hand, needs to be directed to only one group member as the state of the service remains unaltered.

Each group member replies to a `remove` operation. However, the client's runtime system returns only the first arriving member reply (transparent multicast style). If required, clients can access the replies of all group members using a non-transparent multicast style.

An object group request succeeds as long as at least one group member survives the request. The replication degree of an object can be increased at run-time. When an object requests to join a group, the run-time system first suspends group invocations. Then the runtime system requests the internal state[1] of a group member by issuing its `get_state` member function. Subsequently, the runtime transfers the state over the network to the newcomer object and invokes its `set_state` function with the state as a parameter. Finally, group invocations are resumed. The whole procedure is fault-tolerant; if the state-sending member fails, state transfer is restarted with another member.

The underlying group management protocol guarantees a consistent replicated state of the `Directory` even when several clients fire `install` and `remove` operations while a new object is joining the group. This execution style is called *Virtual Synchrony* [2].

4 Applicability

The object group pattern can be very effective when there is a need for fault-tolerance, efficient dissemination of data, load-sharing or a combination thereof. Some interesting application areas are:

- **Fault-tolerant client/server systems**. The proposed pattern supports the implementation of highly available objects. Object groups can provide active replication, passive replication, and multi-versioning.

- **Groupware**. Object groups facilitate the implementation of applications such as teleconferencing, video-on-demand, distributed whiteboards and other kinds of groupware. Information flows between groups of individuals can be modeled and implemented in an efficient manner.

- **Ticker services for financial information**. In such applications, a continuous data stream (for example quotes from a stock exchange) needs to be transmitted to a possibly large set of receivers (for example to trader workstations) in an efficient and reliable manner (Figure 1). The pattern allows the efficient transmission of data over protocols such as IP-multicast. Reliability is ensured by object replication and reliable multicast.

- **Scalable text retrieval systems**. A parallel text retrieval engine can be implemented in the form of an object group with N members (Figure 2). Each group member is assigned a portion of the text database. To perform a search, a client application multicasts the sought text pattern to the group. Each of the N group members needs to search only 1/N of the database. As is explained below, group members have a consistent opinion on the group size in spite of failures and in spite of retriever-objects that join the group at run-time to increase parallelism. For this kind of application, a nearly linear speedup can be achieved by employing the object group pattern.

- **Caching**. In certain situations, the response time of a service is decreased if provided by an object group since replicas of the service can be placed at the sites where the service is frequently accessed. In the extreme case, a replica is created per client application and placed on the clients' machines. A client transmits read-only requests only to the local replica; write-requests, on the other hand, are multicast to all replicas.

- **Network management**. Object groups offer a convenient solution to the problem of collecting and propagating management and monitoring information in distributed applications.

- **Object migration**. Requests are multicast through opaque object references and senders need not know the network addresses of the members. Consequently, an object can leave a group, move to another place and then rejoin the group and continue working. Object groups hence offer support for mobility and for system reconfiguration. This kind of object migration is useful for preventive maintenance and for coarse grained load balancing.

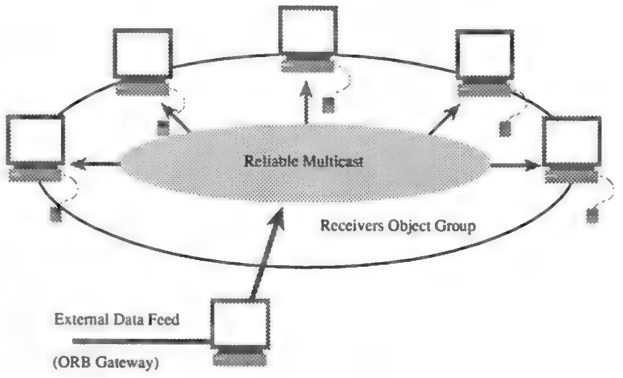


Figure 1: Reliable Stock Exchange Ticker.

5 Virtual Synchrony

The main requirement our pattern makes on the underlying communication subsystem is that it imple-

[1]for example, the entries in an electronic phone book.

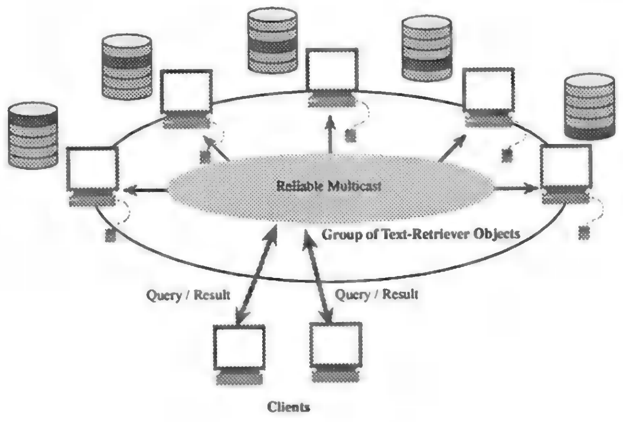


Figure 2: Scalable Text Retrieval Engine.

ments the Virtual Synchrony model. Virtual Synchrony makes possible the run-time replication of mutable objects and the consistent detection of failures. Moreover, the notion of consistent management of group membership information is a fundamental component of the Virtual Synchrony model. The motivation behind Virtual Synchrony is to allow programmers to assume a closely synchronized style of distributed execution, even though the underlying system is asynchronous [2]:

- The execution of an object consists of a sequence of events which may be internal computations, request transmissions, request dispatch, failure notifications or changes to the membership of groups of which the object is a member.
- A global execution of the system consists of a set of object executions. At the global level requests are sent by multicast or unicast.
- Any two objects that receive the same multicasts or observe the same group membership changes see the corresponding local events in the same relative order.
- A multicast to an object group is delivered to its full membership. The send and delivery events appear as a single, instantaneous event.
- However, the model provides only the *illusion* of synchronous execution: Whenever possible, synchronization is relaxed to reduce the degree to which processes can delay one another, and to better exploit the parallelism inherent to an application.

6 Structure and Participants

Figure 3 depicts structure and participants of the object group pattern. Figure 4 shows a possible layering of the communication subsystem.

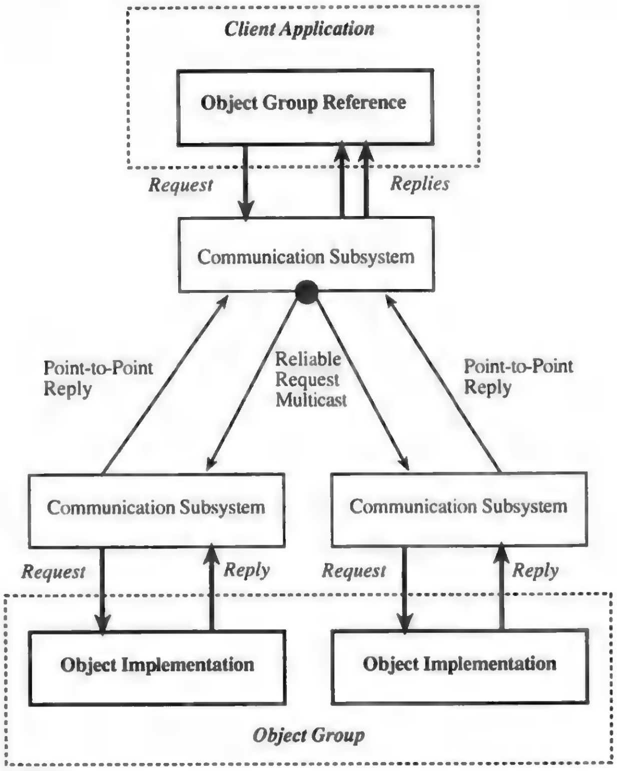


Figure 3: Structure of the Object Group Pattern.

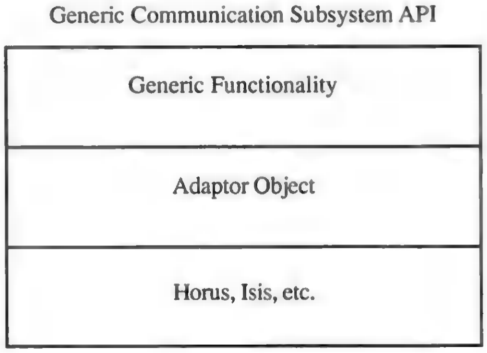


Figure 4: Layers of the Communication Subsystem.

Client Application

The *client application* creates an instance of an object group reference and binds the reference to a target group by using a name server to perform the name-to-object mapping. After successful binding, operations issued through the reference are transmitted to the object group.

Object Group Reference

An *object group reference* is used as a "smart pointer" to an object group. The object group reference is a surrogate object with the same interface as the group members. When a member function is invoked on the group reference, the runtime marshals the arguments, multicasts the request to the group members, waits for one or more member-replies, unmarshals the replies and passes the result back to the client. An object group reference can be used like a conventional CORBA object reference.

Communication Subsystem

The *communication subsystem* (or *runtime*) provides a low-level interface to reliable multicast, group management and light-weight processes. In this paper, we assume that the communication subsystem is part of a CORBA object request broker. We suggest implementing the communication subsystem on top of toolkits such as Horus [19], Isis [2], Consul [12], Phoenix [11], Totem [13], or Transis [1] as these provide low-level system support for process groups, reliable multicast and Virtual Synchrony. For the sake of flexibility and portability, the communication subsystem implements a generic interface. An Adaptor Object is used to map the generic interface onto the real interface provided by the toolkit (Figure 4).

Object Implementation

An object group consists of one or more *object implementations*. An object implementation provides the functionality of the service; for example, it implements a phone directory or a receiver of stock exchange quotes. Each group member implements the same IDL interface.

Object Group

The object implementations form a logical *object group*. The ORB ensures that each group member receives all requests that the client applications have submitted through the object references (reliable multicast). Optionally, the ORB may guarantee that each object implementation dispatches requests in exactly the same order (totally ordered multicast).

Each group member obtains a unique *rank* number. The oldest member is assigned rank 0, the next-oldest obtains rank 1 and so forth. Whenever an object joins or leaves (either explicitly or by crashing). the group, the group members obtain a *view change notification* from the ORB to inform them of their new ranks and of the number of group members.

7 Collaborations

Server Side

If an object implementation wants to join a group, it obtains an object group reference from a name server or by using the CORBA `string_to_object` function. Subsequently, it invokes a `join` function with the reference as a parameter. (There is also a `leave` and a `create_group` function). After calling `join`, its `set_state` member function is invoked by the ORB to update its internal state to the state of the other group members. In the above `Directory` example, the state consists of the `string/any` entries maintained by the service.

Client Side

When an application wants to send a request to an object group, it obtains an object group reference either from a name server or by using the CORBA `string_to_object` function. Now the application can submit requests through the group reference in a type-safe manner. If required, the client application can request that each group member's reply be returned (non-transparent multicast).

View Management

At all times, a member of a group has a certain *view* of which other objects belong to its group. A view is an ordered set of object references; there is one reference per group member. The first entry of a view (`view[0]`) represents the oldest member, the second entry (`view[1]`) the second oldest and so forth. Each group member knows its own index in the view. This ranking is consistent among all group members. Join and leave operations trigger the installation of a new view at each group member's runtime system.

View management is an important component of the Virtual Synchrony model. To hold views consistent, the object group pattern employs a view management protocol similar to the following (Figure 5).

The oldest member is said to be the *coordinator* of the group. When an object wishes to join a group, the object's runtime sends a `JOIN_REQUEST` message to `view[0]` (to the coordinator). The coordinator multicasts a `FLUSH_REQUEST` message to the group members. Upon receipt of a `FLUSH_REQUEST`, a member

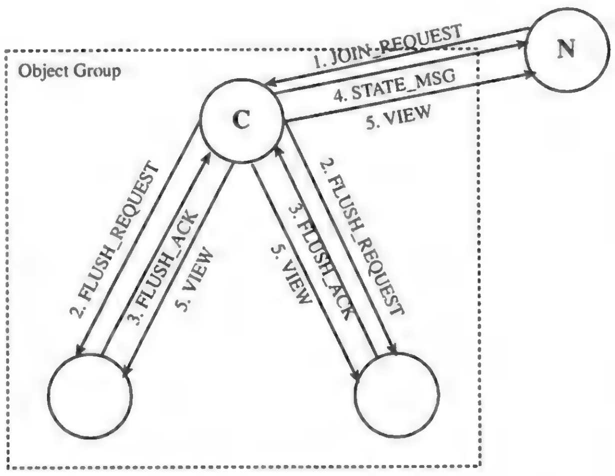


Figure 5: View Management Protocol. Object N requests to join a group containing three objects. Object C is the coordinator of the group.

submits any cached requests it needs to have delivered in the old view. Each member confirms that it is done with the flushing by returning a `FLUSH_ACK` message to the coordinator. A member is not allowed to submit any more requests since the view is now unstable. When the coordinator has received a `FLUSH_ACK` from every surviving member, it sends one or several `STATE_MSG`s containing its internal state to the newcomer object. After this state transfer is completed, the coordinator multicasts a `VIEW` message containing the new view. When a member receives a `VIEW` message, it updates its local `view` vector with the data in the message. The view is stable and the members may continue to submit requests.

If the coordinator fails, the group members receive a failure notification from the runtime and the member with object reference `view[1]` becomes the new coordinator. The view transfer is then restarted with the new coordinator.

The view management mechanisms are mostly transparent to the user of the object group pattern. However, by overloading specific member functions, an application dependent action can be triggered whenever a new view is installed. Variations of this view management protocol are discussed in [17, 15, 7, 4].

8 Consequences

The object group pattern offers the following main benefits:

- *Fault-tolerance* through active replication, passive replication or multi-versioning. This makes possible fault-tolerant client/server systems.
- *Efficient group communication* over Ethernet, IP-multicast and other facilities that enable multicast. The object group pattern provides a high-level, invariable interface to such facilities. This supports the implementation of groupware systems and of certain applications in the financial domain.
- *Load sharing* through a group of objects in which each member performs only a part of the computations associated with a client's requests. This makes possible parallel text retrieval applications and the like.
- *Scalability* through the replication of services which are mainly accessed by read-only operations.
- *System reconfiguration* through dynamic object migration.
- Last but not least, an object-oriented interface to toolkits like Horus and Isis is provided.

The object group pattern has following drawbacks:

- An efficient and robust implementation of the pattern requires *sophisticated system support* not yet available in today's commonly used operating systems. This system support includes: reliable, totally ordered multicast of network messages, a group management protocol to ensure that object group members have consistent views on which objects are in the group, failure detection and consistent propagation of failure notifications.
- Fault-tolerant *name servers* must be provided, allowing objects to retrieve the references of the groups they want to join.
- Fault-tolerance is *not transparent* to the programmer. Programmers have to deal with object groups, with join and with leave requests. For stateful objects, programmers need implement state transfer upcalls unless an IDL compiler is provided to generate state-transfer code out of enriched IDL specifications.

9 Implementation

Group Management API

In a CORBA Object Request Broker, the object group pattern can be implemented *as an extension* of the CORBA Basic Object Adapter (BOA):

```
// C++
class GroupBOA: virtual public BOA {
public:
    // Operations on object groups:
    //
    static void create_group(Object_ptr group,
        const ProtocolPolicy& policy
        =default_protocol_policy);
    void join(Object_ptr group);
    void leave(Object_ptr group);
    static void destroy_group(Object_ptr group);

    virtual void get_state(AnySeq& state,
        Boolean& done);
    virtual void set_state(const AnySeq& state,
        Boolean done);
    virtual void view_change(const View& newView);
};
```

The **create_group** method creates a new object group and binds the object reference **group** to it. The reference can be installed in a name server or converted to a human-readable string by the ORB::object_to_string operation. The **policy** argument tells the underlying toolkit what kind of multicast protocol to employ; for example, total ordering or causal ordering [8, 5]. If the BOA is configured to run on Isis, the **policy** object selects one of the Isis *abcast*, *cbcast*, *fbcast* or *gbcast* protocols. In the Horus configuration, the **policy** object selects a Horus protocol stack[2] [18] . The policy-object mechanism can be extended to cover quality of service guarantees. For example, the minimum bandwidth necessary to sustain a certain service can be defined through a policy object.

Objects in the network join or leave a group by retrieving its reference from a name server and by issuing the join or leave operation with the reference as a parameter. **destroy_group** irrevocably destroys an object group but without destroying the group-members themselves.

State Transfer

When an object joins a non-empty group, the ORB requests the internal state (that is, the values of the instance variables) of some group-member by invoking its **get_state** method. Subsequently, the runtime transfers the state to the newcomer and invokes the newcomer's **set_state** method. The runtime can transfer a large state in fragments by repeatedly invoking the state transfer functions until TRUE is assigned to the done argument of **get_state**. An object-state is represented as a sequence of CORBA any objects.

[2]Horus supports a variety of ordering and reliability protocols, as well as communication over UDP, IP-multicast, ATM, Mach messages, and so forth.

State transfer is necessary for redundant computations to permit the replication-degree of a stateful object to be increased at run-time. The programmer must write application-specific **get_state** and **set_state** methods. These methods can also be used to checkpoint the state of an object to non-volatile storage or to perform object migration.

The **view_change** method of an object is invoked whenever another object joins or leaves the group. The **newView** object contains the new cardinality of the group as well as the object references of the group members.

Call Styles

The programmer may specify how many member-replies the runtime should collect during an object-group invocation. Furthermore, operations that leave the state of an object unchanged can be tagged as *readonly*. Such operations are automatically transmitted by point-to-point communication to only one group member. To help implement this, we suggest that a **readonly** operation attribute be provided by the IDL. Following call styles can be selected by the programmer on a per-call basis:

- **ALL**: This call type demands that the replies of all operational group-members be collected and returned to the caller. The caller is suspended until all group members have replied.
- **MAJORITY**: The call is active only until a majority of the members have replied.
- **ONE**: The call is active only until the first member-reply is received. This call-style is used for transparent multicast.
- N: Any number ranging rom 1 to the number of members of the group can be specified. If the specified number of replies cannot be collected due to a failure, an exception is thrown.

Object Migration

In order to migrate an object implementation that is the only member of a certain group, one has to instantiate a new object implementation on the destination machine and to join the object to the group. The state of the obsolete object will automatically be transferred to the newcomer object and the two objects will be synchronized. The obsolete object can then be destroyed.

10 Sample Code

The following code illustrates the replication of a `Directory` object and the interaction with the resulting `Directory` object group.

Server Side

The code fragment below demonstrates a server application that instantiates an implementation of interface `Directory`, creates an object group if given the `-c` option, and joins the newly created object to the group. The `-c` option also causes a group reference to be installed into the naming service. The `-j` option causes the `Directory` object implementation to join an existing group.

```
// C++
main(int argc, char **argv) {
    Object_var obj, group;
    // a reference to the naming service:
    extern CosNaming::NamingContext_ptr nc;
    // create an implementation of interface Directory:
    _im_directory obj0;
    ...
    switch(argv[1][1]){
    case 'c': // create and join a group:
        // get a reference for obj0:
        obj = obj0.create(rd, pt, dpt);
        // create an empty object group:
        obj0.create_group(obj);
        // install group reference into name server:
        nc->bind("directory", obj);
        // join obj0 to group:
        obj0.join(obj);
        break;
    case 'j': // join an existing group:
        // retrieve group reference from name server:
        group = nc->resolve("directory");
        // join obj0 to group:
        obj0.join(group);
        break;
    default:
        cerr << "usage: " << argv[0] << " -c|-j\n";
        exit(1);
    };
}
```

Client Side

The following code fragment shows a procedure that is part of a client application. The object group reference is retrieved from the name server. Subsequently, a transparent and a non-transparent request is issued through the reference. A non-transparent request is appropriate when the members of a `Directory` group provide different information; for example, one member might maintain business phone numbers, another member home numbers:

```
// C++
void clientProc(){
    extern CosNaming::NamingContext_ptr nc;
    Any entry; AnySeq *entries;
    char *number;
    Environment env; EnvironmentSeq envs;
    directory_var dir =
        Directory::_narrow(nc->resolve("directory"));

    // transparent multicast
    dir->lookup("J. Smith", entry, env);
    entry >>= number;
    cout << "The phone number of J. Smith is "
        << number << "\n";
    ...

    // non-transparent multicast
    dir->lookup("J. Smith", entries, envs);
    for(i =0; i < entries->length(); i++){
        (*entries)[i] >>= number;
        cout << "A phone number of J. Smith is "
            << number << "\n";
    };
    delete entries;
}
```

11 Known Uses

The object group pattern has been implemented in the Electra [9, 10] and in the Orbix+Isis [6] ORB.

Electra

Electra is a flexible CORBA-2 Object Request Broker based on the object group paradigm. In Electra, the object group pattern is implemented as an extension of the CORBA Basic Object Adapter (BOA), as was proposed in Section 9. Electra supports the dynamic replication of CORBA objects, failure detection, asynchronous communication and light-weight processes. The ORB is portable; the present version of Electra runs on both Horus and Isis. Electra is publicly available through the Web at `http://www.olsen.ch/~maffeis/electra.html`.

Orbix+Isis

Orbix+Isis is an adaptation of Orbix from Iona Ltd. that runs on top of the Isis toolkit. To become a member of an object group, an object implementation must inherit from a base class that implements either an Active Replica or an Event Stream group style. The Active Replica object group offers three communication styles: Multicast, Client's Choice, and Coordinator/Cohort. Orbix+Isis is a commercial product available from Isis Inc. For more information contact `info@isis.com`.

12 Related Patterns

Proxy Pattern

A *Proxy* provides a surrogate or placeholder for another object to control access to it [16, 3]. Analogously, an *Object Group Reference* provides a surrogate or placeholder for a group of objects.

Coordinator/Cohort Pattern

Coordinator/Cohort [2, 10] is a form of redundant computation in which only one object (the coordinator) performs the computations associated with the client requests. Several cohort objects are associated with a coordinator, acting as "hot standbys". When the coordinator fails, one of its cohorts takes over.

Coordinator/Cohort is a special form of the object group pattern: coordinator and cohorts make up an object group in which the oldest group member is the coordinator. When the coordinator fails, the cohorts receive a view change notification from the ORB and the oldest cohort becomes the new coordinator. New cohorts can be included at run-time.

Event Channel Pattern

The *Event Channel Pattern* [14] provides the abstraction of a highly available, persistent message bus. The Event Channel allows an object to "post" requests and to "subscribe" for requests it is interested in[3]. Posting and subscription can be by ASCII strings that represent topics of interest.

Objects can post requests for receivers that happen to be unavailable. To that purpose, the Event Channel provides persistency of requests. Receivers may later connect to an Event Channel to retrieve a backlog of requests. Thus, clients are decoupled from servers and communication is asynchronous.

The Event Channel pattern can be seen as a variation of the object group pattern. The patterns differ mainly in that Event Channel allows its members to *selectively* listen only for certain requests. Another difference is that Event Channels allow the *spooling* of requests on non-volatile storage so that requests can be directed to group members that are temporarily unavailable. Object Group, on the other hand, excludes any member that has been unresponsive for a relatively short period of time, typically in the range of 10 to 30 seconds.

[3]similar to Usenet.

Acknowledgements

The author would like to thank Robert Ward for many helpful comments. A preliminary version of this material was discussed at the OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel, and Distributed Object-Oriented Systems. The author would like to thank the workshop participants for their valuable suggestions and for their encouragement.

References

[1] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. Transis: A Communication Sub-System for High Availability. In *22nd International Symposium on Fault-Tolerant Computing* (July 1992), IEEE.

[2] BIRMAN, K. P., AND VAN RENESSE, R., Eds. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[3] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[4] GOLDING, R. A., AND TAYLOR, K. Group Membership in the Epidemic Style. Tech. rep., University of California, Santa Cruz, 1992.

[5] HADZILACOS, V., AND TOUEG, S. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, S. Mullender, Ed., second ed. Addison Wesley, 1993.

[6] ISIS DISTRIBUTED SYSTEMS, INC., IONA TECHNOLOGIES, LTD. *Orbix+Isis Programmer's Guide*, 1995. Document D071-00.

[7] JAHANIAN, F., FAKHOURI, S., AND RAJKUMAR, R. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Princeton, New Jersey, Oct. 1993), IEEE.

[8] LAMPORT, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (July 1978).

[9] MAFFEIS, S. Adding Group Communication and Fault-Tolerance to CORBA. In *Proceedings of the 1995 USENIX Conference on Object-Oriented Technologies* (Monterey, CA, June 1995), USENIX.

[10] MAFFEIS, S. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, University of Zurich, Department of Computer Science, 1995.

[11] MALLOTH, C. P., FELBER, P., SCHIPER, A., AND WILHELM, U. Phoenix: A Toolkit for Building Fault-Tolerant, Distributed Applications in Large Scale. In *IEEE SPDP-7 Workshop on Parallel and Distributed Platforms in Industrial Products* (San Antonio, TX, Oct. 1995), IEEE.

[12] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. *Distributed Systems Engineering Journal 1*, 2 (Dec. 1993).

[13] MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., BUDHIA, R. K., LINGLEY-PAPADOPOULOS, C. A., AND ARCHAMBAULT, T. P. The Totem System. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing* (Pasadena, CA, June 1995).

[14] OBJECT MANAGEMENT GROUP. *Common Object Services Specification Volume I.* OMG Document 94-1-1.

[15] RICCIARDI, A. M. *The Group Membership Problem in Asynchronous Systems.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, Nov. 1992. No. 92-1313.

[16] SHAPIRO, M., ET AL. SOS: An Object-oriented Operating System – Assessment and Perspectives. *Computing Systems 2*, 4 (Dec. 1989).

[17] VAN RENESSE, R. The Horus Uniform Group Interface. Horus Documentation.

[18] VAN RENESSE, R., AND BIRMAN, K. P. Protocol Composition in Horus. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing* (Ottawa, Ontario Canada, Aug. 1995).

[19] VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (Apr. 1996).

Pattern Languages for Handling C++ Resources in an Exception-Safe Way

Harald M. Müller
Siemens AG Austria
harald.mueller@siemens.at

Abstract

Using exception handling in C++ can lead to severe problems with dynamic objects and other resources—dangling pointers, memory leaks etc. By using a small set of patterns (collected into a pattern language "Exception-safe C++ objects"), these problems can be avoided. However, it turns out that these pattern make use of "resource management", which poses a lot of questions in itself. For the solution of these problems, a second, somewhat larger set of patterns (collected into a pattern language "Responsibility management under exception handling") is provided.

1. Introduction

Using exception handling in C++ can lead to severe problems with dynamic objects and other resource—dangling pointers, memory leaks etc. A number of articles have dealt with these problems (e.g. [1], [2], [3], [7]), and some solution patterns have been provided. Here, we collect all these solutions and organize them into a pattern-like way for easy application (and possible criticism).

The article is organized as follows: Section 2 provides three patterns for solving fundamental problems of exception handling. The last of these patterns requires correct resource management for all resources in a program, which is a non-trivial problem in itself. Therefore, section 3 goes on to provide patterns for a generalization of resource management, called responsibility management. Before dealing with "good patterns", a bad one is also shown as an example of how *not* to do it. After that, various patterns for managing responsibilities are provided, depending on the services provided by the environment (e.g., is a garbage collector available or not) and on the complexity of how an object might be managed by other objects (e.g. is there always a single manager for each object?). One important case that requires three patterns of its own occurs if the responsibility for a resource is to be passed around among several managers, yet, there must always be exactly one responsible at any time. The patterns dealing with this case are described in section 3.4.2.

The patterns in this article are presented in the Alexandrian form described in [8]; at a few places, a "motivation" section or an "alternative name" section is included additionally.

2. Exception-safe C++ objects

2.1. Problem space

Exception handling changes the nature of algorithms considerably: When an exception is thrown deep in the call stack, it is possible that many functions are suddenly interrupted in the middle of a computation (indeed, this is the rationale for exception handling). This means that those functions will not be able to fulfill the duty they were designed for, which could mean that data structures are no longer in a usable state. A simple example of such an event is the following: Consider a class for storing Gregorian dates:

```
struct GregorianDate {
    string month;
    unsigned day;
    unsigned year;
};
```

Assume that an object of this type stores the date June 30th, 1996. The following sequence of operations tries to change this to July 31st, 1996:

```
day = 31;
month = "July";
```

If an exception is thrown in the assignment to month (e.g. due to lack of memory) and month keeps its original value "June", we suddenly have stored a bad date—"June 31st, 1996". If the date values are used to index a table later on, it might well be that the program crashes[*].

The problem is most serious with objects that cannot even be deleted after an exception interrupted an operation on them (see the BoundedStack example in section 2.4.1).

[*]If month does not keep its original value, but contains garbage afterwards, the situation is certainly only worse.

The following text presents some patterns that explain how to deal with this sort of problems. The fundamental problem discussed here was presented in [2], the solutions stem from [7].

2.2. Terminology

Resource

A resource is a single object or a tightly coupled group of objects that live longer than a single operation (this means that temporary objects are usually not viewed as resources). A resource usually must be created (or "acquired") and released (or "destroyed") by distinct operations.

An example of a resource consisting of a tightly coupled group of objects is the `argv` and `argc` parameters to the `main()` function.

Remark: Although resources are the basic sort of thing dealt with in this article, we give only a vague definition of this term. The reason is that it is not the case that something in a program is or is not resource, but rather that many things can be *viewed* as resources.

For resources that consist of a single C++ object, we will in the following equate resource release and object destruction. However, there might be resources around that provide different operations for resource release (a simple example are C files, which are released by a call to `fclose()`).

Resource invariant

For most resources, not all combinations of states of sub-resources make correct states of the compound resource. For example, the `GregorianDate` struct defined above might have the value

```
day == 42
month == ""
year == -1
```

However, this is not a valid Gregorian date. The same is true for the state

```
day == 29
month == "February"
year == 1995
```

Obviously, there is a condition restricting the values of the sub-resources so that only those values that "make sense" in the application are possible. We call this condition the "resource invariant".

For `GregorianDate`, this is a condition that connects the value of `day` to the name of the `month` and the value of `year` (because of leap years). Moreover, the date must be larger than October 15th, 1582, the day when Pope Gregor introduced the new calendar in (most of) Italy.

At many places in this article, we will use a general Stack class template to describe problems and solutions to them (the declaration of this template is given in the appendix). A few important parts of the invariant of this class are:

- `nelems` is not larger than the size of the array pointed to by `vec`
- `top <= nelems`
- `top >= 0`

The "State Scale"

As mentioned in the introduction, the big problem with exception handling is that the interruption of a computation may leave some resources in a state not acceptable for following operations. However, the "badness" of the state may be of a varying degree—e.g. it might be so bad that the resource cannot even be released; or it could merely have retained its previous state, which means that it is no longer correct, but certainly still usable. The following defines a partioning of the possible states into different categories; the main discrimination is based on the resource invariant introduced above.

After an operation finishes, the state of a resource accessed and possibly modified by this operation may be in one of the following state classes:

- Consistent state - any state fulfilling the resource invariant.
    - Correct state - as it is defined by the specification of the operation.
    - Incorrect state.
        - Same state - the same state as before the operation (if this is not the correct state).
        - Any consistent state - any other consistent state (e.g. a state like after resource creation).
- Inconsistent state - a state contradicting the class invariant.
    - Shut-down state - an inconsistent state where at least resource release is possible.
    - Very bad state - a state where potentially no operation (not even a resource release) is possible.

Resources in an inconsistent state are also called "damaged resources" or "bad resources" in the following; resources in a consistent state are also called "good resources" or "resources in a good state".

2.4. Patterns

2.4.1. "Ensure Shut-down-ability"

• Problem

An exception might interrupt a function modifying some resource at such a moment that the resource is no longer in a consistent state afterwards. In general, the inconsistent state might be a very bad state, which means that the resource cannot even be destroyed afterwards.

• Forces

1. It is necessary that no resource enters a very bad state, as every resource must eventually be released.

2. Indeed, it is preferable that all resources are as high as possible on the state scale.

3. On the other hand, it is very complicated to write all modification functions in such a way that they leave all resources only in consistent states (see e.g. the GregorianDate example in the introduction).

4. At least *some* operations must be possible on resources even if an exception was thrown during a modification.

• Solution

Design all resources and all modification functions on them in such a way that the resources never enter a very bad state.

Heuristic A: Do not have a value necessary for destruction depend on two or more variables ("do not do computations in destructors").

Heuristic B: Do not store values other than those a destructor expects in an object's attributes (e.g. do not use class members as "temporaries").

• Examples

The `GregorianDate` struct is an example of a resource that is designed according to this principle: The (trivial) destructor will work regardless of the values that have been assigned to the member variables.

An example of a class violating this rule is the `BoundedStack` class declared the appendix—a stack class with a maximum number of elements. Its member variable `current_top` always points one element above the top element, and `current_size` is the number of pushed values. The element counter `current_top` is implemented by a class `BoundedInt` that can only represent a bounded interval of integers. Operations trying to assign a value outside this interval throw an exception.

The destructor of a `BoundedStack` object simply computes the base of the allocated array and then deletes it:

```
template <class T>
BoundedStack<T>::~BoundedStack() throw ()
{
   delete [] (current_top - current_size);
}
```

—however, this contradicts Heuristic A given above. The problems arising from this design are explained in the paragraph on "Design rationales".

• Force resolution

This pattern steers an intermediate way between items 1. and 2. in the forces' list on the one hand, and item 3. on the other.

It might appear that this solution is somewhat weak—one would expect that using exception handling would yield more powerful error handling policies that would also allow to repair damaged resources somehow. However, in most real applications throwing an exception means that all resources modified by a function interrupted by the exception can no longer be used. If the resource stores its information non-redundantly—which is the case for the vast majority of data structures—, it is not possible to reconstruct the correct result, anyway. So shutting down the resource and others that depend on it is typically the only possible solution.

• Design rationale

It is important to notice that by this pattern, we explicitly allow resources to be in bad states. One might wonder whether this is really necessary—after all, isn't it possible to keep resources in some good state all the time, e.g. by always returning to the (necessarily good) state from before the exception? Alas, it turns out that this is equivalent to a complete transaction system with full rollback capabilities—certainly not the right approach for the small resources that are customarily needed in a C++ program. For more information, see [7].

Heuristic A: For two or more variables, any modification must occur as a sequence of modification steps. If an exception is thrown by one of these steps, the compound resource consisting of all the variables might no longer be in a consistent state, hence a value computed from some of the variables might be garbage. An example for this is the following implementation of `BoundedStack::push()`:

```
template <class T>
void BoundedStack<T>::push(const T& e)
// throw (overflow_error, ... T&=const T& ) *
{
    ++current_size;
    *current_top++ = e;
}
```

If the increment of `current_size` throws an exception (which is possible if we try to push more elements than the stack can hold), we end up with an incremented `current_size` value, but `current_top` has not changed, and hence the destructor will call `delete` with an invalid argument.

Heuristic B simply addresses a common misuse of member variables.

- **Consequence**

A consequence (or rather *the* consequence) of this pattern is that resources might be in bad states, i.e., states where no operation except release is possible. This means that other parts of the system that might access the resource in the future must get notice of the state of affairs so that they can avoid these accesses. That this should be done explicitly is codified in the next pattern.

2.4.2. "Hide damaged resources till destruction"

- **Problem**

A modification of a resource might fail. Use of "Ensure Shut-down-ability" means that the resource cannot be used afterwards, but only destroyed. In general, every such resource is part of some other, larger resource[†]. The important question now is: When, and how, is it possible to continue using the larger resource?

- **Forces**

1. Keeping all sub-resources of the larger resource always usable is not feasible (see previous pattern).

2. Throwing away each resource with a single damaged sub-resource might result in shut-down of functionalities that could still be provided.

3. It is very dangerous to rely on the promise of other program parts that the damaged sub-resource will not be accessed until destruction of the whole resource.

- **Solution**

a. If the sub-resource's value is part of the resource's invariant, the complete resource must be viewed as damaged and therefore abandoned (if the sub-resource's value is not consistent in itself, the condition of the larger resource cannot possibly be true).

b. Also if the exception thrown might have left values from the invariant in an inconsistent state (although each single value is still usable), the complete resource must be viewed as damaged and abandoned.

c. If the sub-resource's value is not part of the resource's invariant, explicitly hide the damaged sub-resources from all further accesses *except* those in the destructor of the resource.

- **Examples**

Case a. Assume that we repair the `BoundedStack` class shown in section 2.4.1, e.g. by providing a separate data member pointing to the base of the allocated array. Now look at the `push()` operation: If an exception is thrown by the increment of `current_top`, we must assume that this sub-resource is damaged. Hence, we must abandon the complete `Bounded-Stack` object.

Case b. The `push` operation for the stack class must enlarge the internal array if it is full. This can be accomplished e.g. as follows:

```
template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    if (top == nelems) {
        nelems *= 2;
        T *new_buf = new T[nelems];
        for (unsigned i=0; i<top; i++)
            new_buf[i] = vec[i];
        // ... delete vec
        // ... replace it by new_buf
    }
    // ...
}
```

If one of the assignments in the for loop throws an exception, `nelems` will already have been doubled, but the memory pointed to by `vec` is still the old one.

*For an explanation of this notation, see footnote in the appendix, section A.2.

[†]In the last consequence, we can even view all the resources in a program as a single program-wide "program resource."

Hence the part of the invariant saying that nelems is not larger than the size of the array is no longer true, and the complete stack object has to be abandoned.

Case c. At its end, the push operation has to assign a value to an element of the stack's internal array, e.g.

```
template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    // ...
    vec[top++] = e;
}
```

If the assignment fails, the stack is no longer usable afterwards: The top element may be in a bad state, hence no operation except destruction is possible. However, the value of the top element certainly is not relevant for the functioning of the stack, hence it will not appear in the invariant of a stack resource. Thus, it should be possible to continue the use of the Stack object.

To this end, we must hide the damaged element from further use. A simple remedy is to reduce nelems to top so that all methods (except the destructor!) now believe that there are fewer elements allocated (watch out for the increment of top; it must now be postponed until it is clear that the assignment did work):

```
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    // ...
    try {
        vec[top] = e;
        top++;
    } catch (...) {
        nelems = top;
        throw;
    }
}
```

This course of action might be worthwhile for example when the stack is used for recalling commands that a user typed at some earlier time. For such an application, it might make more sense to lose some input than to shut down the complete user interface*.

- **Force resolution**

The damaged sub-resources are no longer available for any functionality the larger resource provides. However, all functionalities not using the damaged parts are still available.

- **Consequences**

It is a necessity to design explicitly the various "restricted functionalities" of a resource, i.e., which operations are still available after various sub-resources went bad.

*But of course, the user should be told about this event in the exception handler (the catch clause).

2.4.3. "Resource management is necessary"

- **Motivation**

The previous pattern showed what to do if part of a resource entered an inconsistent state. However, certain resources must never even go into an *incorrect* state, as we cannot afford to live without them. For these resources, the sequence of operations manipulating them must *always be correct.*

The most outstanding example of such a resource is the dynamic heap. Here, we require for example that for each successful new operation, a corresponding delete must be executed somewhen later. As exceptions may interrupt statement sequences quite forcefully, it is not immediately clear how this requirement might be fulfilled. If one operates the heap naively in the presence of exception handling, throwing an exception might easily interrupt sequences manipulating the heap, thereby leading to memory leaks and/or double deletes (see [2]).

The pattern "Hide damaged resources till destruction," which promises to deal with damaged parts of a resource, does not work for the global heap: It would require us to "hide" "damaged parts" of the heap (e.g. where a double delete or a memory leak has occurred) from further access. However, this is definitely not possible in C++ (and would probably not make much sense anyway, as the available space in the correctly maintained part of the heap would get smaller and smaller until nothing was left).

The Stack::push() method explained in the previous pattern already showed how code that is correct without exception handling will be incorrect when exceptions might get thrown. It contains a section for replacing the internal buffer by a larger one:

```
if (top == nelems) {
    nelems *= 2;
    T *new_buf = new T[nelems];     (a)
    for (unsigned i=0; i<top; i++)
        new_buf[i] = vec[i];        (b)
    // ... delete vec
```

```
        // ... replace it by new_buf
}
```

As we do not know anything about the parameter type `T`, we must assume that its assignment operation might throw an exception. If this happens in the assignment to `new_buf[i]` at (b), the memory allocated at (a) will leak. This simple example shows that we must put some more effort into managing resources like memory allocated on the dynamic heap.

- **Problem**

In the presence of exceptions, each sequence of function calls can potentially be interrupted at any place, which seems to make guaranteed sequences of function calls impossible. However, for some resources (like the global heap), the sequence of operations applied to them—including the parameters supplied—must always be correct. How can this be accomplished?

- **Forces**

1. Correctness is essential, i.e. the invariants for such resources must be true all the time.

2. Complexity of code should be kept to a minimum.

- **Solution**

Have a strategy of explicitly assigning and transferring *responsibilities* that also works if exceptions are thrown. A *responsibility* here means a duty to execute a certain operation at some time in the future.

- **Examples**

For the most important operation to be executed "at some time in the future," namely resource release, some of the possible strategies include:

- Delegate release responsibilities to a central system (usually called garbage collector). This strategy is usually not adopted in C++ programs.
- Arrange for possibility that memory may be doubly deleted without doing any harm. Also this strategy is usually not adopted in C++ programs.
- Pass responsibility for resource release (especially destruction of objects) around so that there is always exactly one *owner* of the object responsible for the release [6]. This is the standard strategy adopted in C++ programs.

- **Design rationale**

The idea behind this solution is that although exceptions can forcefully interrupt statement sequences (and thereby bypass necessary function calls), they cannot change data that are used to store responsibilities. Hence, explicit responsibility management is less vulnerable to possible exceptions.

It might seem that this "solution" is not more than a general "hint." However, the text tries to balance between the two forces that on the one hand, it should be possible to have different strategies how to enforce correct handling of resources, but on the other hand it should be concrete enough to give some guidelines for evaluating and/or designing such strategies.

In some sense, the solution stresses the fact that it is more important to *have* a strategy than *which* strategy is adopted (as long as it is a correct one).

- **Force resolution**

The forces mentioned in the "Forces" section are not immediately dealt with. The reason is that the various possible strategies have different trade-offs. The sections describing concrete strategies will give more information about these trade-offs.

- **Consequences**

It must be admitted that this pattern creates problems of its own: First, strategies fulfilling the conditions of the solution must be developed. Second, one has to decide *which* strategy should be used. Both problems are non-trivial. Because of this, we will view the *solution* given here as a *problem* in itself in the rest of the article, for whose solution we will again provide a set of patterns. These patterns will explain the standard strategies that can be used for responsibility management under exception handling and provide guidelines for selecting among them.

3. Responsibility management under exception handling

3.1. Problem description

- **Motivation**

See section 2 of this article.

- **Problem**

Have a strategy of explicitly assigning and transferring *responsibilities* that also works if exceptions are thrown. A *responsibility* here means a duty to execute a certain (secure*) operation at some time in the future.

*See the following section on terminology.

• **Forces**

1. Correctness is a non-trivial requirement for these patterns in itself, therefore it is mentioned here.

2. The code complexity necessary to implement the responsibility management logics should be as small as possible. This is mainly due to the requirement that the testability, maintainability, documentability, and understandability of the code should not get worse when exception handling is introduced—after all, this language feature is intended to make it *simpler* to deal with exceptional events in a program.

3. The patterns should allow to write programs as efficient as without special responsibility management.

4. In some cases, it is important that resource release or any other operation passed to a responsibility holder occurs at a specific time; in other cases, it is only necessary *that* the operation be executed. The tradeoffs between the earlier requirement for determinism and the possibility of indeterminism are another force on a possible solution.

5. Sometimes resources have the same lifetime as their creator*, i.e., the responsibility for any operation on them can always reside with the creating block. However, "interesting" resources usually live much longer than their creating block is active—here some sort of responsibility transfer will be necessary.

3.2. Terminology

Secure operation - An operation that does not throw an exception. In C++, secure functions are marked by an empty exception specification `throw()`.

Responsibility - A duty to execute some operation in the future. In this article, we assume that these functions are secure operations. For the most important operation needing responsibility management, namely resource release (in many cases accomplished by a destructor), this is a realistic assumption.

Responsibility holder - An object that has a responsibility. Usually, the operation to be executed is placed in the destructor of the responsibility holder, as the destructor is (almost) certainly bound to be executed at some time (this is a small pattern of its own). [6] uses the term "owner" instead of responsibility holder.

Resource management class - A class explicitly designed for responsibility holders (in contrast to the use of e.g. simple pointers for holding responsibilities). One example of such a class (actually a class template) is the `auto_ptr` type defined in the C++ standard.

Creator of a resource - The first responsibility holder that initially creates a resource.

In this discussion, the creator of a resource is always some block, i.e., some sequence of statements.

In [6], also objects and classes are seen as creators if they get control over the resource "quick enough." In the presence of exception handling, this view could be dangerous. Consider a constructor for some class `X`:

```
X::X() throw (X_problem)
{
   member = new Resource;
   // ... some operations that possibly
   // ... throw an X_problem exception
   // ...
}
```

If one assumes that the allocated `Resource` object is under object responsibility, one would expect that no additional actions are necessary in the constructor to release `member` in the case of an exception. However, this is not true: In C++, the destructor is not executed if an exception is thrown in the constructor; hence, resources allocated in the constructor must also be released there.

Because of this, we assume here that the `Resource` object is placed under the responsibility of the enclosing block—whose responsibility it is to release the resource in case of an exception. In order to avoid a differentiation between constructors and other functions, we adopt the view that the creator is always the block enclosing the resource acquisition.

We will deal in the following not with the general problem of strategies for arbitrary operations, but restrict the patterns to the most prominent operation for which responsibility management is necessary, namely resource release ("resource shut-down," "resource destruction"). However, the following patterns can be used also for other operations.

Most of the following patterns could be part of Tom Cargill's "Localized Ownership" pattern language. However, they differ from the patterns given in [6] in that they explicitly provide exception-safe code patterns—which is the topic of this paper, after all.

*See also in the following section on terminology.

3.3. A Bad Pattern

- **Not a (good) solution**

The following is a possible pattern to deal with the problem of correct resource management: After each resource acquisition, a try block encloses the operations on the resource. If any of the operations fails, an unspecific catch block releases the resource and re-throws the exception so that further handlers get notice of the exception.

- **Example**

```
T *r1 = new T[...];
try {
    // ... do something with r1;
    T *r2 = new T;
    try {
        // ... do something with r1 and r2;
        // etc. for each resource allocated
    } catch (...) {
        delete r2;
        throw;
    }
} catch (...) {
    delete [] r1;
    throw;
}
```

- **Force resolution**

There is only one down-side to this pattern, namely that it has a very high code complexity: Each single resource acquisition leads to a separate nesting level and an additional catch block. However, this single problem makes the pattern virtually unusable for real programs, as the high code complexity entails many other software problems, like poor testability, maintainability, documentability, reuse etc.

For the frequent case of purely local resources, there is an additional complexity introduced, as the resource release has to be doubled:

```
T *local_r = new T[...];
try {
    // ... do something with local_r;

    delete [] local_r;
} catch (...) {
    delete [] local_r;
    throw;
}
```

3.4. Good Patterns

The following patterns for solving the responsibility management problem in the presence of exceptions are grouped into four sections:

- Patterns with a single responsibility holder - these contain "Garbage Collector" and the famous "Resource Acquisition is (Object) Initialization" pattern.
- "Sequence of exclusive responsibility holders" describes how a resource is passed from one responsibility holder to another.
- "Reference counting" and
- "Idempotent operations" give solutions where more than one responsibility holders exists for a single duty.

As all the patterns share the common problem and forces described in section 3.1 above, these are not repeated in the following sections.

3.4.1. Patterns with a single responsibility holder

3.4.1.1. "Garbage Collector"

- **Solution**

A dedicated subsystem is capable of finding and destroying all resources that may be destroyed.

- **Examples**

Concepts of garbage collection for C++ are presented in [11]. There are also a number of commercially available garbage collection systems for C++. Moreover, all languages providing garbage collection are examples of this strategy (however, this is not of much help for C++ users in the current state of affairs).

- **Force resolution**

There are four potential drawbacks to this solution:

1. C++ does not provide a garbage collector in standard implementations, therefore such a system must be acquired or implemented. However, implementing a garbage collector is a large problem in itself.

2. Many garbage collection algorithms can fluctuate widely in their consumption of processing power. For systems with narrow timing conditions, this may be a problem (however, the same can also be true for general heap algorithms, where finding a space for an ob-

ject and/or releasing memory need not take a predictable time).

3. Garbage collection is in general indeterministic, i.e. it is not possible to predict when the resource destruction actually takes place. This can be a problem if the destruction accomplishes different tasks from freeing memory, e.g., closing a window or a file.

4. Garbage collection is confined to freeing allocated memory. Other operations (like closing files, releasing allocated time slots on a communications channel etc.) are usually not handled by a garbage collector.

However, there are two possibilities how to handle such cases with garbage collection:

- The "Garbage Collector" *pattern* can be extended to serve other purposes. One example is the automatic closing of files resp. streams at the end of a C or C++ program, which could be viewed as a collection of "file garbage".
- Another possibility is to map all operations to memory release by placing the operations in a destructor call. When the garbage collector releases the memory at some point in the future, the destructor is executed, and the duty is carried out. However, this needs some sort of garbage collector that knows about C++ objects and their destructors, i.e., it must support some kind of *finalization* (see [12]).

• **Remark**

One could also see the garbage collector as a part of the language and not as a separate, identifiable responsibility holder. Under this view, one would interpret this pattern as a complete abandonment of the resource invariant*.

3.4.1.2. "Resource Acquisition is (Object) Initialization"

• **Solution**

Use an automatic (i.e., stack-based) responsibility holder object for managing the resource in the following way:

(a+†) Allocate a resource by creating a resource management object on the stack.

(b+) Use the resource.

(c) The resource is released by the destructor of the resource management class.

For an extended explanation of this pattern, see e.g. [1].

• **Alternative name**

"Resource Release is Object Destruction"

The alternative name stresses the fact that the release operation is the more problematical one. In my opinion, the pattern should be called by this name to point out its potential usefulness more clearly.

• **Force resolution**

With this solution, the resource cannot be passed outside the creating block, hence this pattern solves only a sub-case of the general problem. However, the patterns presented in the following section can be viewed as a generalization of this pattern.

• **Consequences**

Separate resource management classes are needed for each type of resource that may be necessary. The forthcoming ANSI/ISO C++ standard library will contain a template `auto_ptr<T>` that can be used as a resource management class for a single heap object of class `T`. However, it is lacking a similar class for arrays allocated on the heap (this would only have to call `delete []` instead of `delete` in the destructor). For a possible implementation of such a class, see the `auto_ptr_array` class template in the appendix.

• **Remark**

Some other languages provide direct language means for writing statements that are guaranteed to be executed even if an exception is thrown. Examples include the `valueNowOrOnUnwindDo:` method in Smalltalk or the `finally` clause of Java.

3.4.2. "Sequence of exclusive responsibility holders"

• **Solution**

There is always exactly one responsibility holder for resource release. However, by passing around the responsibility among many responsibility holders, it is possible to keep the resource alive longer than the creating block is active.

*Maybe one should derive another pattern from this: "Keep invariants as few and as simple as possible."

†The + symbol means that this step may throw an exception. Steps without a + must be implemented as secure operations.

- **Force resolution**

This solution is a generalization of the "Resource Release is Object Destruction" pattern. It keeps its efficiency and determinism, however, the code complexity is higher than with all other patterns. Moreover, it produces a lot of problems of its own (these will be dealt with in following sections).

The main problem of this solution is that explicit *responsibility transfers* are necessary. This happens for instance if the creating block transfers its responsibility for some allocated memory to an object before it goes out of scope. We will use these responsibility transfers to ensure that all resources get released exactly once even in the presence of exceptions. However, this makes it necessary that the transfers themselves do *not* throw an exception, i.e., they must be *secure*—otherwise, we could no longer be sure where the responsibilities lie after a failed transfer. In almost all cases, this can be ascertained by moving around only some kind of reference value, not the resources themselves. Such values might be pointers, integral values or certain types of iterators that may be securely copied.

This solution will probably be the standard solution used in C++ programs, although in some respects it might be considered the most difficult one. It certainly places the highest burden among all solutions on the programmer.

3.4.2.1. "Responsibility transfer is swap"

- **Problem**

If a responsibility for a resource is passed from a responsibility holder A to another responsibility holder B, it might be the case that B already is responsible for some other resource. Simply discarding this responsibility is obviously not possible, as we postulated that there must always remain exactly one responsibility holder. What are we going to do with the "superfluous responsibility"?

- **Forces**

Besides the ubiquitous correctness requirement, there are two more forces on solutions for this problem:

1. It should not be required to declare and handle more responsibility holders than absolutely necessary; and

2. the code complexity should be small, as this pattern is only a stepping stone for other patterns.

- **Solution**

Swap the responsibilities between A and B.

- **Examples**

The following is an example where a resource manager called `perm` takes over the responsibility for an object of class `X` from a local resource manager. The `swap_with()` method of class `auto_ptr` (which is *not* defined in the proposed C++ standard) exchanges the internal pointers of the two objects. At first we show the wrong solution to this problem:

```
auto_ptr<X> perm;
// ...
{
    auto_ptr<X> temp;
    temp.reset(new X);
    // ...
    perm.reset(temp.release());
    // wrong: perm might already be
    // responsible for some resource!
}
```

The correct solution exchanges the responsibilities of `perm` and `temp`:

```
auto_ptr<X> perm;
// ...
{
    auto_ptr<X> temp;
    temp.reset(new X);
    // ...
    perm.swap_with(temp);
    // temp's destructor can now take
    // care of the resource for which
    // perm was responsible before, if
    // there was any.
}
```

- **Force resolution**

For once, the forces can be perfectly resolved: The pattern is obviously correct; it certainly needs the minimal number of responsibility holders; and the code complexity is really small.

- **Remark**

Swapping responsibilities does not require that one of the resources must be released afterwards. It is perfectly legal to swap two "active" resources that are both used after the swap.

The following two patterns describe two frequent applications of "Responsibility transfer is swap", namely resource replacement and acquisition of a new resource. They refine a group of "Local Ownership" patterns given in [6] by making them exception-safe.

3.4.2.2. "Resource replacement"

▪ Problem

Replace an old resource under object responsibility by a new one. This problem occurs in many basic operations, e.g. assignment (where the old member values must be replaced by new ones), in containers that automatically adjust their internal storage (e.g. a string class that enlarges its buffer if characters are added; or our running example, the Stack class template), and elsewhere.

▪ Forces

1. The resource replacement must be correct, i.e., for the old resource as well as for the new one there must always be exactly one resource manager.

2. The code complexity should be as small as possible.

▪ Solution

(a+) Allocate a local resource by creating a resource management object on the stack.

(b+) The local resource is used (usually initialized) as necessary.

(c) The responsibilities for the object's resource and the local resource are swapped.

(d) The destructor of the resource management object releases the now local resource (the former object resource).

▪ Examples

The following two examples show the application of this pattern to the assignment and the push operation of the stack class template.

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& s)
// throw (bad_alloc, ...T())
{
    if (this != &s) {
        auto_ptr_array<T>
            new_buf(nelems=s.nelems);    (a+)

        for (unsigned i=0; i<s.top; i++) (b+)
            new_buf[i] = s.vec[i];

        new_buf.swap_with(vec);          (c)
        /* destructor of new_buf */      (d)
    }
    return *this;
}


template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    if (top == nelems) {
        nelems *= 2;
        auto_ptr_array<T> new_buf(nelems); (a+)
        for (unsigned i=0; i<top; i++) (b+)
            new_buf[i] = vec[i];
        new_buf.swap_with(vec);          (c)
        /* destructor of new_buf */      (d)
    }
    vec[top++] = e;
    // Instead, we could copy the code from
    // 2.4.2 if it should be possible to use
    // the stack object after an exception in
    // the assignment.
}
```

▪ Force resolution

By applying the "Resource transfer is swap" pattern, the pattern maintains the invariant that there is always exactly one responsibility holder for each resource release.

The complexity of the code is about the same as with the "Resource Acquisition is (Object) Initialization" pattern (in contrast to the try/catch-pattern of section 3.3).

▪ Alternative name

"Two-Step"

This name captures the fact that resource replacement (and also acquisition of a new resource—see the following pattern) always has two phases:

- a first "exception-prone" phase where the resources are acquired and used;
- a second phase of secure responsibility transfers and secure resource releases.

In many cases, operations on resources can be reordered to fit with this two-phase pattern.

However, under special circumstances, it might happen that an exception-prone operation must occur *after* the second phase. A prominent example of this is a `return` statement executing a copy constructor that might throw an exception. In such cases, somewhat more complicated patterns are necessary. For a detailed description of such a problem and its solution, see [7].

3.4.2.3. "Resource acquisition for an object is resource replacement"

▪ Problem

Allocating a new resource and putting it under object responsibility: This is the standard problem when writing a constructor.

▪ Forces

The forces are the same as for the resource replacement problem, i.e. correctness and minimal code complexity.

▪ Solution

(a+) The object's responsibility holder is initialized to an arbitrary resource.

(b+) A local resource is allocated by creating a resource management object on the stack.

(c+) The local resource is used (usually initialized) as necessary.

(d) The resource is placed under the object's responsibility.

(e) The destructor of the resource management object releases the arbitrary resource allocated in (a+)—this is in effect a no-op.

▪ Example

The following example shows the application of this pattern to the default constructor of the stack class template.

```
template <class T>
Stack<T>::Stack()
// throw (bad_alloc, ...T())
   : vec(0)                            (a+)
{
   auto_ptr_array<T> buf(nelems=10);  (b+)
             // (c+ not necessary here)
   buf.swap_with(vec);                 (d)
   /* destructor of buf */             (e)
}
```

▪ Force resolution

See "Resource Replacement" pattern.

▪ Design rationale

One could design a separate pattern for acquisition of new resources that does not need the "arbitrary resource" from step (a). However, the pattern above has the advantage that it inherits its correctness from the "Resource replacement" pattern.

Moreover, this pattern fits well with the C++ style rule that member variables should be set in the initializer list of a constructor: As shown above, the member gets the responsibility for a trivial resource in the initializer list, which is then replaced by the real resource in the constructor's body.

The patterns described in sections 3.4.1. and 3.4.2. assume that there is only one responsibility holder. This has the advantage that it is always clear who will carry out the required operation, but it has the disadvantage that passing around responsibilities is either not possible ("Resource Release is Object Destruction") or quite complex ("Sequence of exclusive responsibility holders"), or that the underlying system must be quite powerful ("Garbage Collector").

The following patterns describe alternative approaches to the responsibility management problem by allowing more than one responsibility holder. From the viewpoint of exception handling, the fundamental problem of correct resource release thereby vanishes (almost): We simply give the responsibility to release a resource to a lot of responsibility holders; if the activities of one of these are interrupted by an exception, we can nevertheless be quite certain that some other responsibility holder will eventually carry out the release operation.

3.4.3. "Reference counting"

▪ Solution

Many responsibility holders take over the job to release a resource. The release happens when the last responsibility holder declares that it wants to do its duty.

For details on this well-known pattern, see e.g. [9] or [6]. It is also mentioned as a special case of the "Proxy" pattern in [10].

3.4.4. "Idempotent operations"

▪ Solution

Make resource release an idempotent operation, i.e. an operation that has no effect after the first execution. Thus, many responsibility holders may exist at one time, each of whom might carry out the release operation.

- **Examples**

An application has a central log file that is to be closed either if the user explicitly requires it, or when the application finishes. For this, a central log-file-handler could provide a function

```
Log_file_handler::close_log_file() {
    delete p_log_file;
    p_log_file = 0;
}
```

The duty to call this function is on the one hand given to a central application shut-down process. However, other processes—e.g. the afore-mentioned user command for explicitly closing the logfile—may equally call this function without giving notice to the shut-down process, as a repeated execution does no harm.

- **Force resolution**

In contrast to the reference counting scheme, with "Idempotent operations," the *first* responsibility holder that wants to carry out its duty will accomplish the resource release. It depends on the application whether this is possible or not.

Moreover, it must be still ascertained that at least one responsibility holder carries out its duty (as with reference counting).

The last problem can be overcome by putting the responsibility to a single central responsibility holder which swears to fulfilling its duties—see "Garbage Collector." We've come a full circle.

4. Conclusion

This article tried to cast a set of rules necessary for writing exception-safe code into patterns form. Widespread use of exception handling, especially in larger systems, is not yet common in the C++ community, therefore it is hoped that patterns of this kind help to deal with the problems inherent in this complex feature.

However, there are some open points that have to be addressed in the future:

At some places, implicit assumptions are not yet clearly spelled out (e.g. that it is assumed that responsibility management objects are not class members, but only local variables); at other places, explicit conditions could be weakened or left away (e.g. there seem to be some meta-patterns behind the patterns presented in 3.5.2).

Moreover, as always, the patterns certtainly need refinement by potential and actual users.

The patterns shown in section 3 are useful for resources that are designed according to "Ensure Shut-down-ability", i.e., it is not necessary that the resource itself is in some good state after an exception occurred. For resources that must have a quality as high as e.g. the global heap, i.e., where it must be possible to continue using them after an exception interrupted some operation, more powerful patterns are necessary that guard *all* modifications of all variables in a piece of code. An introduction to concepts for solving this problem can be found in [7]; shaping them into pattern form might be the contents of a future article.

5. References

[1] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, Reading, MA, 1990.

[2] Cargill, T. Exception handling: A false sense of security. C++ Report 6(9):21-24, Nov.-Dec. 1994.

[3] Carroll, M. and M. A. Ellis. Tradeoffs of exceptions. C++ Report 7(3):12-16, March-April 1995.

[4] Ellis, M. A. and B. Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA, 1990.

[5] Working Paper for the C++ standard, April 1995.

[6] Cargill, T. Localized Ownership: Managing Dynamic Objects in C++. Proceedings of the 2nd Pattern Languages of Programming Conference, September, 1995.

[7] Müller, H.M. 10 Rules for Handling Exception Handling Successfully, C++ Report 8(1):22-36, Jan. 1996.

[8] Coplien, J.O. Software Design Patterns: Common Questions and Answers. URL: ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps.

[9] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, 1992.

[10] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[11] Boehm, H.J. Garbage Collection in an Uncooperative Environment. Software Practice and Experience 18:807-820, Sept. 1988.

[12] Boehm, H.J. A garbage collector for C and C++. URL: ftp://parcftp.xerox.com/pub/gc/gc.html.

A. Appendix

A.1. Declaration of GregorianDate

```
struct GregorianDate {
    string month;
    unsigned day;
    unsigned year;
};
```

A.2. Declaration of Stack<T>

```
template <class T>
class Stack {
public:
    class XPopEmpty:public domain_error {
    public:
        XPopEmpty(const string& info)
                : domain_error(info) { }
    };

    Stack(); // throw (bad_alloc, ...T() * )

    Stack(const Stack<T>&);
      // throw (bad_alloc,...T(),...T&=const T&)
    ~Stack() throw ();
    Stack<T>& operator=(const Stack<T>&);
      // throw (bad_alloc,...T(),...T&=const T&)
    unsigned count() const throw()
      { return top; }
    void push(const T& e);
      // throw (bad_alloc,...T(),...T&=const T&)
    T pop();
      // throw (XPopEmpty,...T(const T&) )
private:
    unsigned nelems;
    unsigned top;
    T* vec;
};
```

A.3. Declaration of BoundedInt and BoundedStack<T>

```
class BoundedInt {
public:
    BoundedInt(int bottom, int top) throw ()
        : bottom_(bottom)
        , top_(top)
        { }
    BoundedInt operator++()
        throw (overflow_error)
        {   if (++value_ >= top_)
                throw overflow_error(...);
            return *this;
        }
    operator int() throw()
        { return value_; }
    // ...
private:
    const int bottom_, top_;
    int value_;
};

template <class T>
class BoundedStack {
public:
    BoundedStack(unsigned maxsize)
      // throw (bad_alloc, ...T() )
        : current_size(0, maxsize)
        , current_top (new T[maxsize])
        { }
    ~BoundedStack() throw ();
    void push(const T& e);
      // throw (overflow_error,... T&=const T& )
    // ...
private:
    BoundedInt current_size;
    T* current_top;
};
```

A.4. Declaration of auto_ptr_array<T>

```
template <class T>
class auto_ptr_array {
public:
    auto_ptr_array(size_t n)
        // throw (bad_alloc, ...T())
        : ptr_(new T[n]) { }
    ~auto_ptr_array() throw ()
        { delete [] ptr_; }
    T& operator[](unsigned i) throw ()
        { return ptr_[i]; }
    T* release() throw ()
        {   T* tmp = ptr_;
            ptr_ = 0;
            return tmp;
        }
    void swap_with(T*& q) throw ()
        {   T* tmp = ptr_;
            ptr_ = q;
            q = tmp;
        }
private:
    T* ptr_;
};
```

*The ...-notation is used to designate the set of all exceptions possibly thrown by a certain sort of expressions. Here it means "all exceptions possibly thrown by calling a parameterless constructor of T." This special sort of comment is used as C++ has no facility to "inherit" throw specifications from other functions.

The Any Framework
A Pragmatic Approach to Flexibility

Kai-Uwe Mätzel, Walter Bischofberger
UBILAB
Union Bank of Switzerland
{maetzel,bischofberger}@ubilab.ubs.ch

Abstract

During the development of Beyond-Sniff, a distributed multi-user development platform, we were confronted with various, apparently unrelated problems: data, control, and user interface integration of distributed components, system configuration, user specific preferences, etc. Undoubtedly, it is not trivial to find solutions for such issues, but C++ makes it even more challenging due to its static nature and insufficient meta-information. To overcome these shortcomings, we implemented a small and powerful framework called Any[1]. The Any framework augments C++ with a flexible, dynamic, garbage-collected data representation mechanism. It serves as a language-independent data integration vehicle and provides data management and declarative retrieval facilities.

1 Introduction

In 1991, we began to develop the C++ programming environment Sniff [2]. Motivation for this project was that framework-centered software development greatly increases the requirements for development environments [3],[5]. Furthermore, there were no programming environments that scaled and provided the extensive browsing support we needed.

After Sniff was successfully commercialized, we started to work on Beyond-Sniff [4], a platform and set of tools for co-operative software engineering. The goals of this project are to develop a conceptual framework for co-operative software engineering, and to build the development environment needed for its enactment.

A software engineering environment provides support for so many different activities that a monolithic design would make no sense. This is especially true for co-operative software engineering environments, where the environment also provides communication and coordination support. Modern co-operative software engineering environments consist of a number of integrated tools. In the ideal case their integration is so seamless that the user believes him- or herself to be working with a single tool.

1. The name clash of our Any framework with the CORBA any data type is coincidental and does not imply a similarity.

A Beyond-Sniff environment consists of a set of tools, which use several shared services in order to provide functionality, as depicted in Fig. 1. These tools are relatively lightweight because much of their functionality is already implemented in the shared services.

Beyond-Sniff has the following characteristics which are especially relevant to this paper:

- It is a distributed environment built with a focus on scalability.
- It provides data, control and user interface integration mechanisms [15].
- New tools and services can be integrated at runtime, thus making their functionality immediately available.
- Tools and services can be substituted. Whether tools or services are substitutable depends upon their capabilities - not upon their implementation details, e. g. the implementation language used.
- A user can tailor his or her environment without affecting other users.

Beyond-Sniff has been designed and implemented using object technology, especially framework technology. Starting points were the application framework ET++ [17], C++, and the current implementation of the Sniff programming environment running on various flavors of the UNIX operating system. We decided to start under these conditions because they allow us to profit from years of experience as well as the results of our former work. Further information about Beyond-Sniff can be found in [4].

1.1 What are the problems we faced?

During the development of Beyond-Sniff we were faced with the following problems:

Data integration and message exchange between heterogeneous, distributed components. All components of Beyond-Sniff (tools and services) have to be able to exchange data amongst each other. Whether the data is exchanged or physically shared depends on its size and

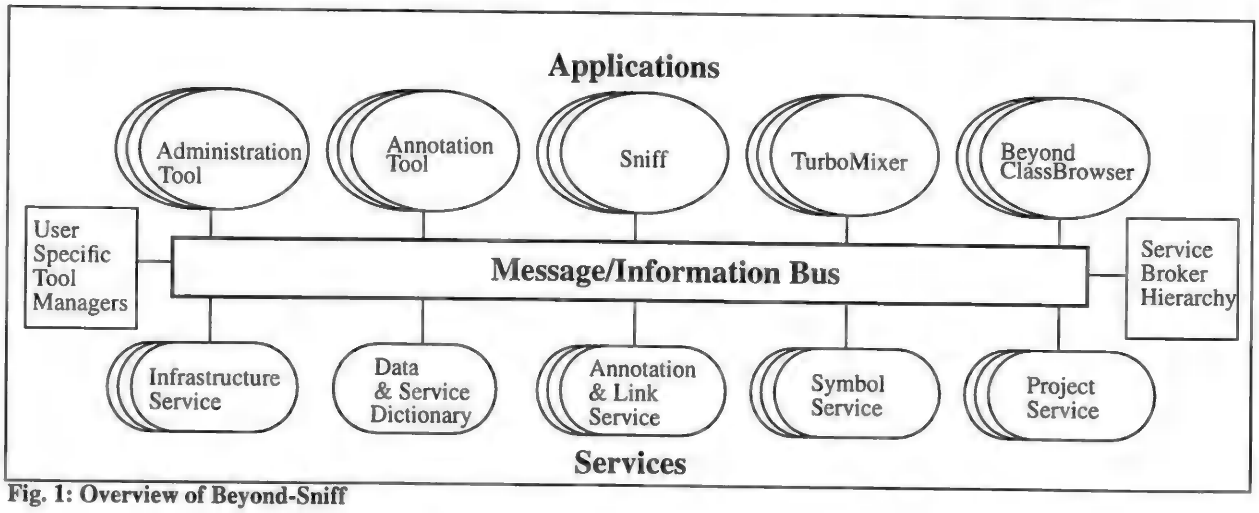


Fig. 1: Overview of Beyond-Sniff

the purpose of its use. In both cases, all components involved must agree on the meaning of the data. Its format cannot be predefined due to the openness of Beyond-Sniff. Furthermore, it should be possible to exchange data even if the receiver knows the sender's data model only partially and different languages are used.

Declarative data access. If large amounts of data have to be shared and therefore made accessible, the access should be declarative (e. g. through pattern matching or query languages) and scalable. This makes it impractical to use the data representation system of the host language.

User interface integration mechanism. Integrating a set of services and tools in a way that they provide the same experience as a stand-alone tool.

Reconfiguration and extension of open extensible environments. If new components can be integrated at runtime, they must be able to inform the rest of the system about their capabilities. Obviously, capabilities cannot be predefined. But to make the new functionality accessible for each user (for instance by means of a new menu entry), tool or service, all system entities must be ready to deal with them.

Schema evolution. The system has to be capable to handle various data model versions of a certain service or tool.

User specific tailoring. Tailoring of a user's environment requires a sophisticated preference system. Again, the format cannot be predefined because the adjustment features of future tools cannot be predicted. Preferences have to be persistent beyond session boundaries.

All these problems are not specific to Beyond-Sniff. They are common to a wide range of large systems. In order to tackle these problems, the following requirements have to fulfilled. They are independent from a specific system or application, especially independent from Beyond-Sniff.

R1 a language-independent, extensible, self-describing (i. e., semantic, according to Sims [16]) data representation mechanism, which can be smoothly integrated with arbitrary programming languages,

R2 fast, change-tolerant, alphanumeric, and binary IO mechanism,

R3 declarative or rule-based data access,

R4 persistent data storage, and

R5 robustness due to explicit data modeling and runtime type checking.

If a semantic data representation, a mechanism that fulfills R1, additionally meets R2, e. g. the listed I/O features, it can be used as a sophisticated message exchange format. This is crucial especially for distributed systems, since it eliminates the difference between external and internal data representation. A mechanism that meets all requirements R1 - R5 is called extended semantic data representation mechanism.

Obviously, there are systems that provide some of the listed features (namely certain dynamically typed languages and 4GLs). But there is no working system under our given external constraints[2] providing all of them. Therefore, we have had to invent a solution that meets the requirements as closely as possible and fits well into our development environment.

Note, that the external constrains do not reduce the applicability of our mechanism but rather makes it a

2. These are the implementation language C++ and that the mechanism has to be integrable with the existing code base of Sniff.

Anything usage

```
Anything employees, anEmployee, address;

anEmployee["LastName"]= "Weinand";
anEmployee["FirstName"]= "André";

address["City"]= "Mountain View";
address["Street"]= "High School Way";
address["State"]= "CA";
address["ZIP"]= 94041;

anEmployee["Address"]= address;

employees.Add(anEmployee);
employees.Add(anotherEmployee);
```

Anything import/export format

```
# array with 2 elements
        { # 1st array element; a dictionary
                /LastName "Weinand"
                /FirstName "André"
                /Address {
                        /City "Mountain View"
                        /Street "High School Way"
                        /State "CA"
                        /ZIP 94041
                }
        }
        { # 2nd array element
                /LastName ...   ...
        }
}
```

Fig. 2: Anything examples

generic solution which could be very useful in many other cases.

Before stepping into the Any Framework, we sketch some related work to show our sources of influence and inspiration. Then, we describe design and implementation of the Any Framework - our approach to an extended semantic data representation mechanism.

2 Related work

The presented systems and mechanisms are selected according to their contributions and ideas to meet the given requirements R1 - R5.

2.1 "Anythings"

André Weinand introduced a semantic data representation mechanism called Anythings. He was motivated by the following problems he had to solve with C++:

Dynamic Extensibility. In object-oriented systems, many objects of the same class exist playing slightly different roles in the context where they are used. These roles require the various objects to carry slightly different information. Usually, this results either in the definition of a large number of classes, which actually should belong to the same type, or in the definition of a comprehensive set of data members, which is used only partially by most instances.

The first approach is unacceptable because it results in an explosion of the number of classes. The second approach wastes a lot of memory. Both approaches make the system harder to understand and force the objects to carry only predictable information. But frequently, developers are faced with problems where this is too inflexible. Instances should therefore be dynamically extensible with arbitrary information.

An interesting application of extensible objects is information "piggybacking". The different parts of a framework are coupled through a sophisticated information flow that usually involves dozens of mediators. Assuming that different parts want to extend the information they exchange, then the originally transmitted object only has to be extended with that information. This does not affect any of the mediators.

Attaching arbitrary information reveals the necessity for a semantic data representation mechanism (R1) because this information has to be interpretable for the receiver.

Streaming and configuration. Similar problems as with information piggybacking occur with object streaming. In order to rebuild an object from a stream the knowledge of how to do so is required. Due to information hiding, this is usually encapsulated in the object's class. Therefore, the creator needs access to this code. To encode the object into a semantic data representation, would free the stream mediators from their dependence on this code. Note, that this actually decouples the receiver of such a stream from its source. If additionally the stream format is human-readable, it can be used for debugging purposes as well as a simple but flexible configuration or preferences mechanism.

Data structure mining. Legacy information systems sometimes export data in more-or-less structured formats. The problem is to parse the format and to dynamically construct a data type, whose instances represent the different results in a uniform way. The evolved data type could be for instance a class in which the set of data members correspond exactly with the set of occurred keys.

Anythings were designed to support solutions for the above problems. An Anything example is shown in Fig. 2. Their most important attributes:

- scalar data types such as integers, strings etc.,
- composed data types, e. g. arrays and dictionaries,
- automatic conversion between all scalar types, and from scalar types to arrays,
- meta-information about the structure of dictionaries,
- garbage-collection based on reference counting,
- human readable streaming format.

These data types are mainly derived from languages like AWK or Perl which incorporate highly flexible data structures like arbitrary nested associative arrays.

Anythings are highly flexible and well suited for a many tasks in small-scale development. They meet R1 and partially R2. However, their high convertibility violates R5 and makes them dangerous to use as an exchange mechanism in large distributed systems like Beyond-Sniff. There, the components have to negotiate and use a common data model. The data itself, not the access operation, should decide about the type of the accessed data.

Furthermore, to represent entire data models it is necessary to have additionally higher level data types, such as classes., A type system that provides classes and as much of the Anything's flexibility as possible, proved to be better than the pure Anythings. Such a system is just as applicable as Anythings - but safer, because it meets R5.

Anythings were not only the inspiration for the Any Framework, but were also our first code base.

2.2 Smalltalk Meta-Classes

Reflection is indispensable for a semantic data representation mechanism. It is a prerequisite for dynamic type checks, related operations, and therefore the best way to make the data representation self-describing and robust (R1, R5). Reflection can be found in many dynamically and statically typed languages or systems. Usually, the implementation of these features is quite similar to their realization in Smalltalk [10]. For each structural entity there is an object describing it.

2.3 NewtonScript

Beside Anythings, we consider NewtonScript another source of inspiration, especially in consideration to the requirements R3 and R4. NewtonScript is the prototype-based object-oriented programming language for the Apple Newton. The basic entities in NewtonScript are frames. A frame consists of various named slots which can hold either a scalar or composed data item or a functional block known as method. NewtonScript introduced a structuring concept called soups.

"... soups are persistent storage objects that hold collections of related data items called entries, which are frames ..." [1].

Soups do not define an internal structure, they only serve as a grouping mechanism. They are object pools and may not be recursive.

Soups are very useful in their role as natural scopes for retrieving particular frames or objects, which is of great importance for R3. Some systems like ET++ already have a technically quite similar mechanism frequently called the "object table". In contrast to NewtonScript's soups, the object tables are usually used as a hidden, internal book-keeping mechanism to implement reflection and not as a grouping concept.

2.4 ODMG Object Database Standard

The ODMG Object Database Standard defines an intuitive object query language. We found the declarative aspects (R3) of OQL [7] convincing. It is well designed for class based object systems and is based on the proven concepts of a query language.

2.5 CORBA and RPC

RPC [6] and CORBA [13] are widely accepted and used. These techniques prove a good fit for distributed systems solving exactly defined tasks. However, their design does not take adequately into consideration the facts that systems have to exchange large amounts of arbitrary structured data and that they evolve over time.

Evolution. With both techniques, the developer implementing the changes has to rebuild even those clients that do not use the new extensions or are not affected by the changes of their servers. This shows that this kind of static interface definition is not as evolution-friendly as we expect it to be for Beyond-Sniff, or generally, large distributed systems.

In contrast to RPC, CORBA reveals less resistance to changes, but only if either

- interfaces are regularly extended by means of inheritance or
- the IDL data type "any"[3] is used.

Necessary changes can be realized by interface inheritance only in few cases. More often, many small changes have to be made, for example few signatures

3. IDL-any is used for the data type any in IDL-scripts as well as for their related structures in the actual implementation language as defined by the CORBA standard.

have to be changed at just one position of the parameter list.

Using IDL-any, the signature changes could be encapsulated by shifting changes from the type level to the runtime structure of data items. Numerous approaches are possible: a) each parameter list consists just of one IDL-any, b) each parameter list includes an IDL-any at its last position, c) only parameter lists which are intended to be extensible include an IDL-any. Changes can then be performed entirely on the structure of the IDL-any, which does not affect the original signature.

Data exchange. A related problem to evolution is data exchange. The physical exchange of structured data is limited within CORBA to the IDL data types or, within RPC, to the data types provided by the RPC's data representation mechanisms like XDR. In many cases this does not satisfy the needs of the applications built on top of them. IDL-any can be used to encapsulate the actual exchanged data structure.

Problems with IDL-any: Evolution as well as data exchange can be tackled by means of IDL-any. But in both cases there are several risks. Followingly, we describe two problems of the application of IDL-any.

- In large systems the IDL-any can only encapsulate an extended semantic data representation mechanism because of its lack of abstraction and flexibility. Why?

  An IDL-any instance can represent an arbitrary data graph. The leafs are either IDL's basic types, constructed types, templated types, or arrays.

  The CORBA standard defines just the basic operations needed to deal with such data graphs. There are no general mechanisms for comparing, sub-graph matching, iteration and similar operations. Furthermore, there is no concept of structural equivalence of particular data graphs, which reveals the lack of means for classification. This implies that each IDL-any has to be individually treated and that the basic operations have to be implemented by everybody, who intends to use IDL-any intensively.

  Beside these lacks of abstraction IDL-any is completely limited to such data graphs. There is no abstraction for dynamic data structures like dictionaries. Although there are ways[4] to emulate dynamic data structures, the usage of these constructions would rely on various conventions and therefore be clumsy to use.

  IDL-any meets requirement R1 only partially because of its lack of extensibility, whereas it meets R2 almost completely (except the alphanumeric representation). I/O mechanisms are intrinsics of the concrete CORBA implementations. Furthermore, the intended usage of IDL-any is limited to CORBA applications which makes it hard to use in other applications.

- The CORBA-defined broker algorithm bases on statically defined interfaces. Signature changes have therefore strong impact on the actual behavior of this algorithm. This is no longer true as soon as IDL-any are used to encapsulate signature evolution.

  Under this point of view, the integration of an extended semantic data representation mechanism with IDL-any can only be a pragmatic solution. In the long term we consider it indispensable to turn IDL-any into a semantic data representation mechanism itself. It should meet at least R1, R2, and R5 and considered accordingly by the object broker algorithms.

4. An IDL-any could be used to represent a particular value of a dictionary. It cannot be used to represent the concept dictionary. This would be possible through a combination of CORBA objects, which would represent the concept and use IDL-any to transfer values between them.

2.6 NEWI

NEWI [11] is an integration environment for co-operative business objects (CBOs). It has to cope with some of the problems listed above: Flexible data and control integration between distributed, heterogenous components. It tackles the problems in a manner quite similar to Anythings. The NEWI equivalent to Anythings are so called Semantic Data Streams (SDS). NEWI prefers semantic data streams instead of interface definition languages such as CORBA-IDL, for reasons similar to the problems mentioned in section 2.5 [16].

3 The design of the Any Framework

In the following chapter we present Anys, our design of an extended semantic data representation mechanism, that has the features (R1-R5) we presented above. Some implementation details of the various language bindings follow. The usability of Anys will be demonstrated with two real world examples.

3.1 Introduction

The Any Framework comprises scalar and compound data types. Additionally, it contains a object-based data type called AnyFrames. Object-based means that AnyFrames do not support methods. The structure of an AnyFrame is defined by its class. Single inheritance between classes is supported. Frames are rather similar to objects, except that they do not have attached functionality. Frames can be grouped by AnySoups.

```
AnyContext *gContext;

void InitSymtab() {
        AnyFrameDesc value(gContext, "AnyNodeValue",
                        "Description",          new AnySlotDesc("AnyString"),
                        "PropertyList",         new AnySlotDesc("AnyDict"),
                        "Deletable",            new AnySlotDesc("AnyBool"),
                        0);
        AnyFrameDesc node(gContext, "AnyTree",
                        "Name",         new AnySlotDesc("AnyString", eSingleValue | eMustHave),
                        "Value",        new AnySlotDesc("AnyNodeValue"),
                        "Children",     new AnySlotDesc("AnyTree", eMayNotRemove),
                        0);
}

AnyFrame MakeTree() {
        AnyFrame rt(gContext, "AnyTree", "Name", new AnyString("Root"), 0);
        AnyFrame n1(gContext, "AnyTree", "Name", new AnyString("Node1"), 0);
        AnyFrame n2(gContext, "AnyTree", "Name", new AnyString("Node2"), 0);


        AnyDict pl;
        pl.Append("A", 1);
        pl.Append("B", 2);
        pl.Append("C", 3);

        AnyFrame v(gContext, "AnyNodeValue",
                "Description", new AnyString("Value of Node 1"),
                "PropertyList", &pl,
                "Deletable", new AnyBool(TRUE),
                0);

        rt.Append("Children",n1);
        rt.Append("Children",n2);
        n1.Append("Value", v);
        n2.Append("Value", v);

        return rt;
}

main() {
        gContext= new AnyContext();
        InitSymtab();
        AnyFrame tree= MakeTree();
        cerr << "The name of the root node is " << tree.At("Name").AsString() << "\n";
        PrettyAnyWriter().WriteAny(cout, tree.ContAt("Children"));
}
```

Fig. 3: Any code example

Soups provide declarative retrieval facilities as appropriate means to describe sets of frames. Each soup can manage an arbitrary number of indices. Indices are used to control and speed up frame retrieval. They are well known from database technology.

Beyond all the features listed in section 1.1, we gave Anys a few more to make their usage more convenient:

- Anys are very flexible and dynamic. At lifetime, the relations between them usually tend to be so complex that it would require tremendous efforts from a programmer to care about their memory management. Therefore, Anys are garbage-collected. This enhances their fit with languages like Smalltalk and makes the C++ programmer's life easier.
- Anys are closely integrated into the host language.

Smooth bi-directional conversation facilities between the host language's data world and the Any data world exist. Each Any implementation has to provide a comprehensive set of built-in transformations.

- Furthermore, it should be possible to consider data items of the host language's world as parts of the Any world without having to transform them in advance. For instance, it should be possible to plug a C++ object into an Any. This requirement conflicts with language independence. A pragmatic solution is that the plug, like the C++ object, is only visible inside the Any for the component that created it. It can be neither streamed out nor streamed in.

3.2 Remark to the C++ code examples

In order to illustrate the definition, the text includes concrete C++ code examples. To understand these examples some knowledge about the C++ implementation of Anys is indispensable. Anys are implemented according to the handle/body class idiom; more exactly, the reference counting idiom. In [8] Coplien explains: "Use of two (or potentially more) classes where an instance of one serves as a manager for instances of the other is called the handle/body class idiom. ... When the body class contains a reference count manipulated by the handle class, such use is called reference counting idiom." The Bridge Pattern [9] describes a more general form of the handle/body idiom. The code examples only show the handle class.

3.3 Basics

3.3.1 Scalar and composed types

Anys comprise the following scalar data types: AnyBool, AnyInt, AnyDouble, AnyString, AnyProxy, and FlatAnys:

- AnyStrings are symbols in Smalltalk terminology. If several AnyStrings have the same value then they refer to the same string. Indeed, strings exist only once.
- AnyProxies are used to bind any kind of object or data of the host language's world into a composed Any.
- FlatAnys are specialized AnyStrings. FlatAnys represent composed Anys that are streamed into a string. This makes it possible to partially delay the reconstruction of Anys when reading them from a stream. This is crucial, if large amount of data are transferred. Furthermore, comparing two FlatAnys checks the structural equality of the represented composed Anys.

The composed data types are AnyArray and AnyDict:

- An AnyArray is a flexibly growing array of Anys.
- An AnyDict, is an array of slots addressed with AnyStrings, containing zero, one, or more Anys. Inserting an Any at a slot which does not yet exist results in its creation.

3.3.2 Frames

AnyFrames are like specially typed AnyDicts. The types define classes over the structure of the AnyFrames. A class defines the names of all slots, their types, properties, and values, as well as their number and sequence.

In the Any system, classes are represented by AnyFrameDescs, an abbreviation for AnyFrame descriptor. An AnyFrameDesc is a sequence of AnySlotDescs. The explicit representation of classes and their structure makes it possible to extend the type set or certain types at runtime and to dynamically check the type of a frame.

Runtime errors occur when accessing an AnyFrame at a non-existent slot or inserting an Any at a slot for which no AnySlotDesc exists in the frame's AnyFrameDesc.

AnyFrameDescs are shared between AnyFrames within a soup or context. Soups and contexts are described in section 3.4.1. Fig. 3 shows a simple and instructive Any example.

3.3.3 Semantic streaming

Anys have only a very limited built-in streaming facility. The produced stream does not ensure that a structurally identical Any can be reconstructed. If the case demands it, the semantic streaming facilities provided by AnyWriters and AnyReaders must be exploited. An AnyWriter is a visitor [9] that travels over the structure of a given Any and writes a self-describing output format onto a given stream. Consequently, an adequate AnyReader can reconstruct an identical Any from that stream.

The following reader/writer pairs must be provided by an implementation:

- schema-tolerant, ASCII-based reader/writer,
- schema-intolerant, ASCII-based reader/writer,
- binary-reader/writer.

A schema-tolerant, ASCII-based reader/writer pair is the ideal instrument for semantic streaming. The writer also streams the complete meta-information, e. g. all necessary type information and AnyFrameDescs. The reader can then reconstruct even AnyFrames whose classes were previously unknown to it.

Schema-intolerant reader/writer pairs include the basic type information but not the AnyFrameDesc. If a reader is forced to rebuild an AnyFrame without knowing its AnyFrameDesc, this results in a runtime error as opposed to the alternative of constructing an AnyDict. Fig. 4 shows the semantic streaming output of the schema-tolerant, ASCII-based PrettyAnyWriter of Fig. 3.

```
//Any:PrettyAnyIO
[2
  <
    "AnyTree"
    "Name"
     [1 "Node1" ]
    "Value"
     [1
       <
         "AnyNodeValue"
         "Description"
          [1 "Value of Node 1" ]
         "PropertyList"
          [1
            {
              "A"
               [1 1 ]
              "B"
               [1 2 ]
              "C"
               [1 3 ]
            } &10
          ]
         "Deletable"
          [1 T ]
       > &5
     ]
    "Children"
     [0 ]
  > &0
  <
    "AnyTree"
    "Name"
     [1 "Node2" ]
    "Value"
     [1 *5
     ]
    "Children"
     [0 ]
  > &20
]
```

Fig. 4: Semantic Streaming

3.4 Advanced features

Additional features are necessary to sensibly use Anys, especially AnyFrames. These features are soups and a declarative data access within soups.

3.4.1 Soups

Soups have been introduced mainly for two reasons:

- An easy to use utility to structure AnyFrames into disjunct, retrievable, changeable sets is considered indispensable. AnyArrays could serve a similar purpose with the drawback that the programmer has to insert each AnyFrame manually. An AnyFrame should be inserted automatically into a given soup at the point of its creation and removed at destruction time.
- Soups in the Any system play the same role as name spaces in C++. All AnyFrames of a soup share their AnyFrameDescs, which are only valid in their soup. This mechanism enables the use of efficient, schema-intolerant AnyReader/Writer pairs. Assume that all AnyFrameDescs between two soups are identical. It is then sufficient to transmit the raw data of an AnyFrame without additional meta-information.

The two purposes of soups are so fundamental that we defined two abstractions. An AnyContext maintains a table of AnyFrameDescs. Therefore, an AnyFrame can only be created within an AnyContext to be able to locate its AnyFrameDesc. An AnySoup, which is derived from AnyContext, additionally manages a collection of AnyFrames and provides query based retrieval and locking support on it.

3.4.2 Querying and indexing

AnySoups support declarative access to their AnyFrames. In contrast to navigating access, declarative access allows a set of AnyFrames to be described. It avoids the need to iterate over all of them and to check if they match a certain pattern. AnySoups incorporate an OQL query processor [7] which can evaluate any OQL query and delivers an AnyArray with all the matching AnyFrames. We chose OQL because it is an object-oriented standardized query language that matches well with our AnyFrame concept.

Query evaluation can only be performed reasonably fast on large AnySoups if they support comprehensive indexing. Indices are managed by AnyIndexManagers. Each soup can have an arbitrary configurable set of index managers.

An implementation has to provide at least the following index managers:

- the OnlyOneIndexManager, which actually provides no index, or with other words all frames are in the same index,
- the StandardIndexManager, which manages a separate index for each AnyFrame type.

Fig. 5 demonstrates two OQL queries. Note, that slots in AnyDicts as well as in AnyFrames are multi-valued. To access all values at once, the All method has to be used. By convention, ALL_<name> is always the name of an index.

```
# Returns an array of AnyDicts containing class name/metho
# name-pairs for all classes except class Any and all
# "Impl"-classes where the method names are not Get or Set.

select  struct(ClassName: x.Name,MethodName: y.Name)
from
        x in ALL_Class,
        y in x.All("MethodDef")
where
        x.Name != "Any"
        and not x.Name.Match("Impl")
        and y.Name.Match("^[GS]et")

# Returns type and name of all classes, method
# implementations, and includes

define symbols as
(       select  struct( type: "Class", Name :sym.Name)
        from    sym in file.All("Class")
) + (   select  struct( type: "MethImpl", Name :sym.Name)
        from    sym in file.All("MethodImpl")
) + (   select  struct( type: "Include", Name :incl.Name)
        from    incl in file.All("Include"),
);
select  symbols
from    file in ALL_SymtabFile
```

Fig. 5: An OQL-query for Anys

3.5 Discussion of language binding problems

Implementations of Anys exist in C++, Smalltalk and Python. This allows us to exchange Anys between programs written in these languages.

In C++ as well as in Smalltalk, we developed frameworks to implement Anys. Different tasks require different degrees of exploitation of a used mechanism. To make the use of a solution always profitable, the interface has to provide various layers of abstraction. Frameworks are an appropriate tool to realize this.

Obviously, the design of, and the effort needed to implement, these frameworks depend upon the underlying platform; just have a look at garbage collection! In C++ the reference counting idiom had to be implemented. The handle and body classes and all assignment operators had to be implemented. In Smalltalk, garbage collection is for free.

3.5.1 Smalltalk

We shall shortly present the two main alternatives to integrate Anys with Smalltalk. We had the choice of implementing Anys according to their definition, or to integrate them seamlessly with the language. The excellent dynamic features of Smalltalk would make this possible.

All scalar data types as well as AnyArray and AnyDicts could be mapped to Smalltalk intrinsics. For each AnyFrame type, the system could dynamically generate a structurally equvivalent Smalltalk class. In order to be able to distinguish these classes from ordinary Smalltalk classes, which is necessary for streaming, all these classes had to be derived from a single special root class. This seems to be a promising approach, but raises various problems.

If class definitions are changed at runtime, which has to be possible with Anys, then maybe large parts of the application have to be recompiled.

The programmer is no longer conscious that he or she is working with data that is possibly shared between various components in a distributed system. While designing and programming with Anys, we found it very helpful to be constantly reminded of the different purposes of Anys and ordinary data. The programmer is much more aware of the fact that the components exchange messages between each other.

4 Application of the Any Framework

Next, we will demonstrate the usage of Anys by the example of the implementation of Beyond-Sniff's service architecture. First, we describe how Anys ease the implementation of an infrastructure for distributed services. Second, we show how we used Anys and Python [14] to make services more flexible and adaptable at runtime.

4.1 Beyond-Sniff's service framework

Anys by themselves are only an extended data representation mechanism. In order to make them sensibly applicable in the intended context of distributed systems, they must be embedded in a proper infrastructure.

As depicted in Fig. 1, Beyond-Sniff follows a service architecture. All services, independent of their actual task, share common characteristics. They provide synchronous and asynchronous point-to-point or multicast communication and request handling facilities. Consequently, messages are encoded as AnyFrames, which include the transmitted information as well as the usual information like sender, receiver, sequence number, and so on. This information of course is also an

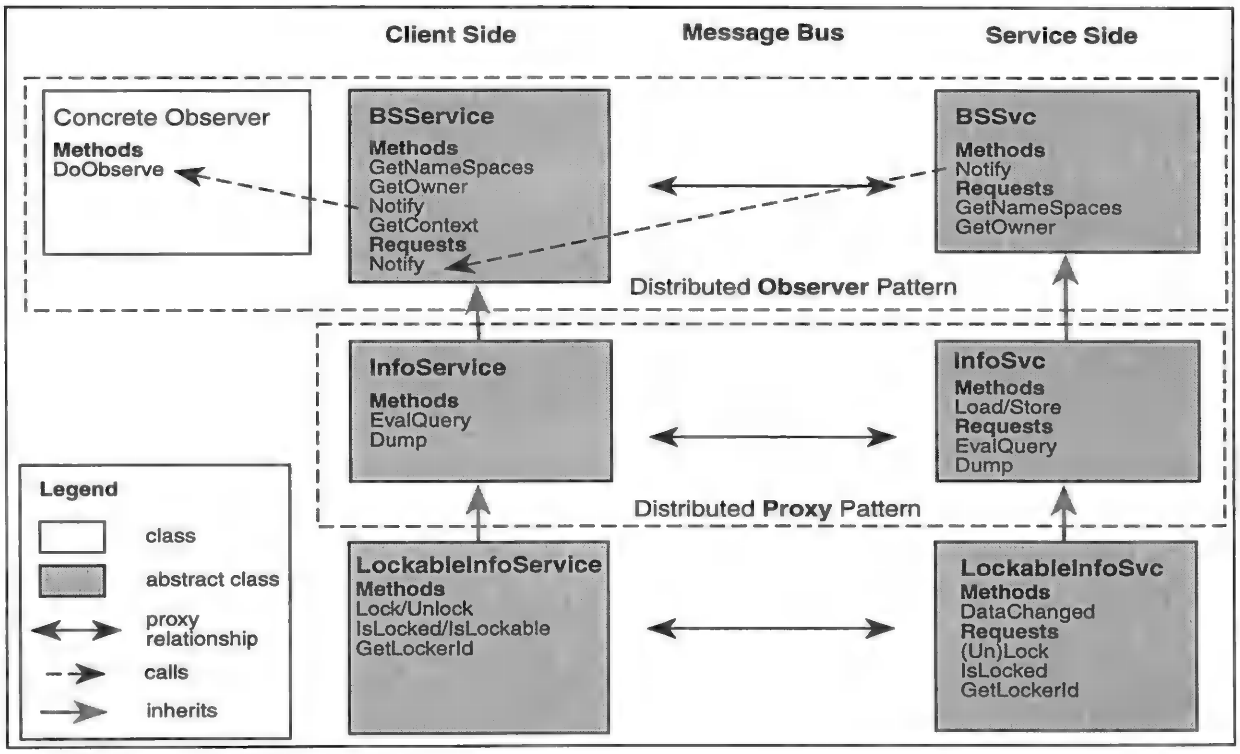


Fig. 6: The Beyond-Sniff client-service framework

Any. NEWI [11] uses a rather similar mechanism to make co-operative business objects (CBOs) communicate language-independently.

Clients and services have to communicate efficiently. Therefore, they negotiate the actual communication context in respect to their capabilities and technical environment. The context mainly consists of a common data model and reader/writer pair that appears to be most efficient.

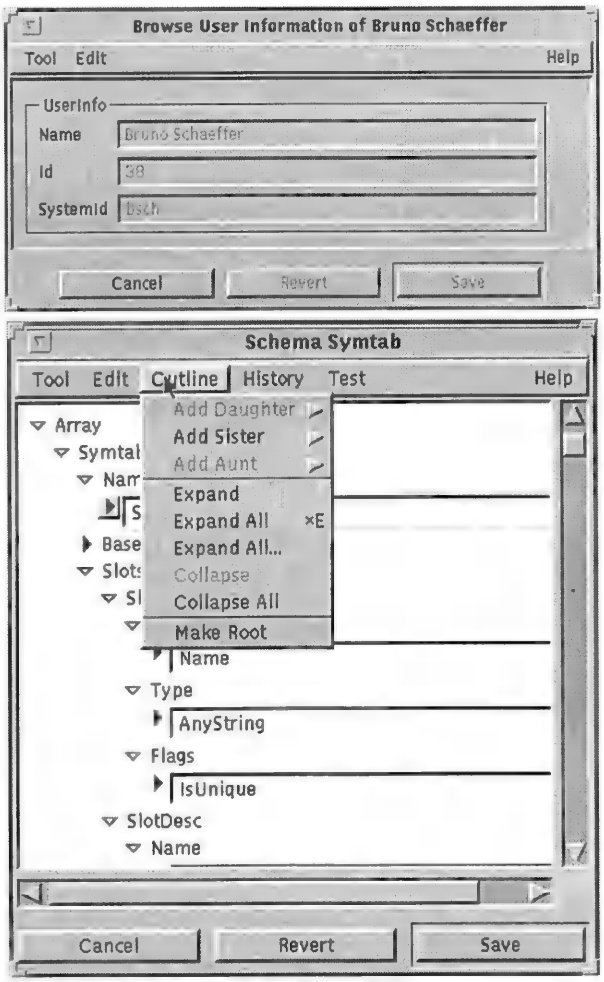
The Any message stream format is determined by the kind of underlying computer architecture of the involved parties. They use binary streaming if both are sitting on top of the same architecture. Beside the negotiated reader/writer pair, more descriptive pairs can always be used. If a message in a more tolerant format arrives, the receiver dynamically selects an appropriate reader.

Data model negotiation is rather simple. The client downloads the server's AnyContext, e. g. its set of AnyFrameDescs, and uses it to create service requests. The service is committed to serve such requests. Furthermore, a client can send any arbitrary request which might not be provided by the server. In this "trial and error" case, the client has to use schema-tolerant streaming.

Service functionality can be invoked dynamically by sending a request directly to the service or by using a service proxy. From the client's point of view, the proxy provides the service although it is just a front end to the real service. This mechanism is provided by a service framework in C++, depicted in Fig. 6. Additionally, it provides indispensable functionality for a distributed infrastructure such as service management, including finding and launching of appropriate providers, monitoring, load balancing, and more.

So far, the service framework solely exploits the basics of Anys. AnySoups with their querying facilities make large-scale data sharing and integration feasible without major effort. Therefore, we introduced information services. An information service manages an AnySoup and lets its clients manipulate and ask queries about the soup's content. In the service framework we provide different abstractions to the concept of an information service, as depicted in Fig. 6.

Visualizing and interactive editing of Anys is almost trivial due to their reflective nature. We exploit this to build generic message-monitoring tools and information service inspectors. Each message can be graphically displayed and manipulated, if necessary. The same is true for the soup managed by an information service. Fig. 7 shows the Generic AnyOutliner and the Generic AnyFrameEditor.


Browse User Information of Bruno Schaeffer

UserInfo

Name: Bruno Schaeffer

Id: 38

SystemId: bsch

Schema Symtab

Fig. 7: AnyOutliner and AnyFrameEditor

The AnyFrameEditor can also be used to create any kind of form-based input dialog cheaply. The data structure to be filled up by using the form, has to be modeled only as a particular AnyFrame. The frame's content can be manipulated with the AnyFrame Editor, which dynamically creates the needed form.

4.2 Introducing scriptability into services

Services, especially in a software development environment like Beyond-Sniff, should be highly flexible and adaptable at runtime. Consequently, services have to have a reflection and a behavior manipulation component. The latter is responsible for providing the demanded flexibility. This can be achieved most comfortably by incorporating a scripting language with the services. We chose Python for this purpose.

The integration of Python with our C++ service framework was fairly straightforward. We extended the service interface with two new requests: EvalScript and InstallRequestHandler. EvalScript is used to evaluate a given script in the receiver's context. With InstallRequestHandler the authorized client can overwrite a particular request handler of a service with a Python script. For information about the integration of Python and C++ consult [14].

As already mentioned, Anys work as a data integration mechanism between different languages. The integration of Anys into Python enables the scripts to work with the data of the C++ service, such as the AnySoup of an information service. Fig. 8 shows this.

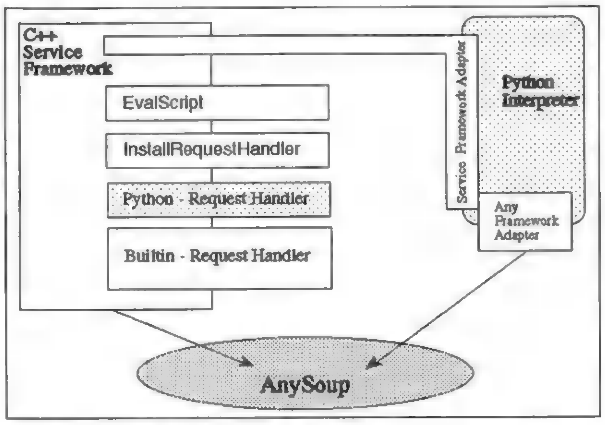


Fig. 8: A scriptable Beyond-Sniff service

5 Experience and conclusions

Anys are a language-independent semantic data representation mechanism which provides semantic streaming, soups, and OQL-based data access. They help the programmer to combine the strength of statically-typed languages like C++ with the convenience of dynamic, extensible, and object-based data types. Anys are widely applicable. Undoubtedly, they are most profitable in the context of distributed systems and language integration. We presented this by the example of the Beyond-Sniff service framework, written in C++, as well as the integration of Python into those services. Both examples represent heavily used mechanisms which are crucial to the Beyond-Sniff project. We can state that Anys proved their usefulness and applicability in large-scale development.

Anys do not provide facilities to attach functionality to certain slots as NewtonScript or Self do [18]. In the context of language-independent data integration, this would allow the introduction of data encapsulation and information hiding. Currently, either each client has to know how the provided data must be used to compute certain functions, or their results have to be part of the data. Function-slots would avoid such redundancies. Although we consider this feature useful, we decided not to implement it, since the integration of a scripting language has sufficiently met our requirements.

Further work on Anys will have to be done in the field of internationalization. Probably, we will provide two different AnyString implementations; one working with unicode characters, the other supporting character code pages.

6 Acknowledgment

We would like to thank André Weinand for his ideas, work, and discussions about Anythings and Anys, as well as for his valuable contributions to this paper. Furthermore, we thank Bruno Schäffer, Michael Scharf and Alexander Schwab for their helpful input. Bruno implemented AnyOutLiner and AnyEditor. Michael and Alexander worked on the integration of Python with the Any and the Beyond-Sniff service framework.

7 References

[1] Apple Computer Inc.: The NewtonScript Programming Languages: PIE Technical Publications, 1993

[2] Bischofberger,W.R.: Sniff - A Pragmatic Approach to a C++ Programming Environment. in Proc. USENIX C++ Conference, Portland, Oregon, Aug. 1992.

[3] Bischofberger,W.R., Kofler,T., Schaeffer,B.: Object-Oriented Programming Environments: Requirements and Approaches. In Software - Concepts and Tools, Vol. 15 No. 2, Springer Verlag, 1994

[4] Bischofberger,W.R., Kofler,T., Maetzel, K.-U., Schaeffer,B.: Computer Supported Cooperative Software Engineering with Beyond-Sniff. In Proc. Software Engineering Environments 1995, IEEE Computer Society Press, Los Alamitos, California, April 1995.

[5] Bischofberger,W.R., Kleinferchner,C.F., Maetzel, K.-U.: Evolving a Programming Environment Into a Cooperative Software Engineering Environment. In Proceedings of CONSEG'95, Tata McGraw-Hill, New Dehli, February 95

[6] Bloomer J: Power Programming with RPC; O'Reilly & Associates, Sebastopol, 1991

[7] Cattell, R.G.G. (Ed.): The Object Database Standard: ODMG - 93. Morgan Kaufman Publishers, San Mateo, California, 1994.

[8] Coplien J: Advanced C++ Programming Styles and Idioms, Addison-Wesley Publishing Company, Reading Massachusetts, 1992

[9] Gamma E, et. al.:Design Pattern - Elements of Reusable Object Oriented Software. Addison-Wesley Publishing Company, Reading, Massachusetts, 1994

[10] Goldberg A., Robson D.: Smalltalk-80 The Language and its Implementation. Addison-Wesley Publishing Company, Reading Massachusetts, 1983.

[11] Integrated Object Systems Limited: Introducing Newi: Integrated Object Manual, Berkshire, 1994

[12] McKeehan J, Rhodes N: Programing for the Newton - Software Development with NewtonScript. AP Professional, Boston, 1994

[13] Object Management Group: The Common Object Request Broker: Architecture and Specification, 1992

[14] Rossum v. G.: Extending and Embedding the Python Interpreter, Stichting Mathematisch Centrum, Amsterdam 1995

[15] Schefstöm D, van den Broek G: Tool Integration–Environments and Frameworks. John Wiley & Sons, 1993

[16] Sims O.: Business Objects - Delivering cooperative objects for client-server; McGraw-Hill Book Company, London, 1994

[17] Weinand, A., Gamma, E.: ET++ - a Portable, Homogenous Class Library and Application Framework. Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, Sept. 1994. Universitaetsverlag Konstanz, Konstanz, 1994, pp. 66-92

[18] Ungar D., Smith R. B.: The power of simplicity; Proceedings of the OOPSLA '87 Conference, Orlando, 1987

Design and Performance of an Object-Oriented Framework for High-Speed Electronic Medical Imaging

Irfan Pyarali
irfan@wuerl.wustl.edu
Eastman Kodak Company
Dallas, Texas, 75240

Timothy H. Harrison and Douglas C. Schmidt
Department of Computer Science
Washington University, St. Louis, Missouri, 63130 [1]

Abstract

This paper describes the design and performance of an object-oriented communication framework being developed by the Health Imaging division of Eastman Kodak and the Electronic Radiology Laboratory at Washington University School of Medicine. The framework is designed to meet the demands of next-generation electronic medical imaging systems, which must transfer extremely large quantities of data efficiently and flexibly in a distributed environment. A novel aspect of this framework is its seamless integration of flexible high-level CORBA distributed object computing middleware with efficient low-level socket network programming mechanisms. In the paper, we outline the design goals and software architecture of our framework, describe how we resolved design challenges, and illustrate the performance of the framework over high-speed ATM networks.

1 Introduction

The demand for distributed electronic medical imaging systems (EMISs) is driven by technological advances and economic necessity [1]. Recent advances in high-speed networks and hierarchical storage management provide the technological infrastructure needed to build large-scale distributed, performance-sensitive EMISs. Consolidating independent hospitals into integrated health care delivery systems to control costs provides the economic incentive for such systems.

Two key requirements for the communication infrastructure in a distributed EMIS are *flexibility* and *performance*. An EMIS must be flexible in order to transfer many types of message-oriented and stream-oriented data (such as HL7, DICOM, and domain-specific objects) across local and wide area networks. EMIS requirements for flexibility motivate the use of distributed object computing middleware such as CORBA [2] in the communication infrastructure. CORBA automates common network programming tasks (such as object selection, location, and activation, as well as parameter marshalling and framing), thereby enhancing application flexibility.

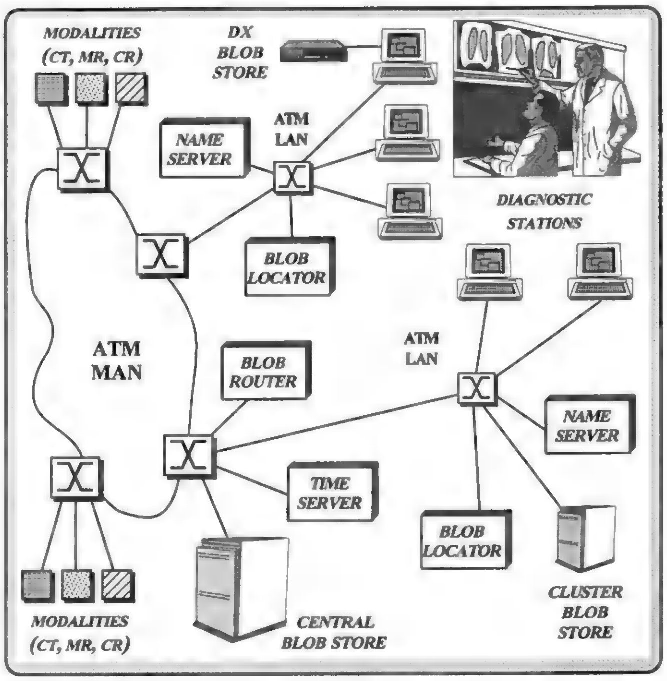


Figure 1: Topology of Distributed Objects in Project Spectrum

However, empirical studies [3, 4, 5] reveal that for bulk data transfer, the performance overhead of widely used CORBA implementations on high-speed ATM networks is 25% to 70% below that achievable using lower-level transport layer interfaces such as sockets or TLI. As high-speed networks like ATM, FDDI, and 100 Mbps Fast-Ethernet become ubiquitous, this performance overhead will force programmers to use lower-level mechanisms to achieve the necessary transfer rates, rather than adopting distributed object computing technologies. This is particularly problematic for performance-intensive application domains like medical imaging, where the use of low-level tools increases development effort and reduces system reliability and flexibility.

To address this problem, we have developed an object-oriented communication software framework called "Blob

[1]This research is supported in part by the Eastman Kodak Company Health Imaging division and the Electronic Radiology Laboratory at Washington University, St. Louis.

Streaming"[2] The Blob Streaming framework is designed to meet the requirements of next-generation electronic medical imaging systems (EMISs). Figure 1 illustrates the topology of our distributed EMIS environment [1]. In this environment, various types of modalities (such as CT, MR, and CR) capture patient images and transfer them as Blobs to an appropriate storage management system (called a Blob Store). Radiologists use diagnostic workstations to retrieve these images for viewing and interpretation. In addition to medical images, next-generation EMISs must support multimedia Blobs such as video streams and audio diagnostic reports.

The Blob Streaming framework provides a uniform interface that enables EMIS developers to flexibly and efficiently operate on multiple types of Blobs located throughout a large-scale health delivery system. This framework combines the flexibility of high-level distributed object computing middleware (*e.g.*, CORBA) with the efficiency of lower-level transport mechanisms (*e.g.*, sockets). Developers of EMIS communication software have traditionally had to choose between (1) high-performance, lower-level interfaces provided by sockets or (2) less efficient, higher-level interfaces provided by communication frameworks like CORBA. Blob Streaming represents a midpoint in the solution space. It improves the correctness, programming simplicity, portability, and reusability of performance-sensitive EMIS communication software. Blob Streaming leverages the flexibility of CORBA, while its performance remains competitive with applications programmed to the socket level.

This paper is organized as follows: Section 2 motivates the design of the Blob Streaming framework, outlines the key design challenges, and describes how we resolved these challenges; Section 3 illustrates how the Blob Streaming framework has been used to build high-performance image transfer applications; Section 4 compares the performance of Blob Streaming with alternative C, C++, and CORBA approaches over a high-speed ATM network; Section 5 discusses recommendations based on our results; and Section 6 presents concluding remarks.

2 Design of the Blob Streaming Framework

2.1 Blob Streaming Architecture

The Blob Streaming framework is designed to minimize excessive layering to improve performance, while still allowing applications to be decoupled from communication details that are prone to change. This decoupling helps increase portability and enables transparent optimizations without altering public Blob Streaming interfaces. The shaded portion of Figure 2 illustrates the architecture of the Blob Streaming framework, which consists of the following layers:

- **C++ wrapper layer:** This layer uses an existing toolkit of C++ wrappers [6] that shield applications from the details

[2]Blob stands for "Binary Large OBject."

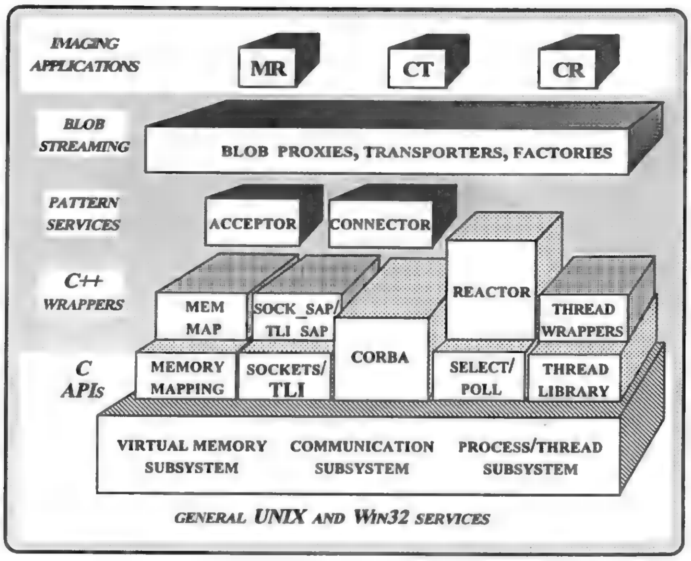


Figure 2: Layering Architecture of the Blob Streaming Framework

of the lower layer C library and OS system call mechanisms. These mechanisms include sockets and CORBA for interprocess communication, memory-mapped file wrappers for optimized secondary storage access, and event demultiplexing. The use of C++ wrappers provides strongly typed interfaces that simplify the development of Blob Streaming. For example, porting to alternative platforms requires no changes to Blob Streaming software because the Blob Streaming library does not directly access any OS specific interfaces. Currently, Blob Streaming is implemented on many versions of UNIX, as well as Win32 platforms.

- **Pattern services layer:** This layer uses an existing framework of strategic design patterns [7] that enhance framework quality by addressing fundamental challenges in communication systems. For instance, Blob Streaming uses the Acceptor and Connector patterns [8] that decouple the passive and active initialization of services from the tasks performed once the services are initialized. Likewise, the Reactor pattern [9] simplifies event-driven applications by associating event handler objects to the demultiplexing of events. The use of these patterns in Blob Streaming leverages prior design efforts and reduces software development risks.

- **Blob Streaming layer:** This layer provides application developers with the Blob Streaming components that provide generic interfaces for high-speed Blob transfer. The main components include Blob *Proxies*, *Transporters*, and *Factories*:

  - *Proxies* – which use the Bridge and Proxy patterns [10] to represent location- and type-independent handles to Blobs. These patterns provide a surrogate that shields clients from knowledge of where the Blob resides, thereby making it easy to vary the location without affecting client code.

- *Transporters* – which use the Strategy pattern [10] to represent location- and type-independent algorithms that perform optimal transfer of Blobs between sources and destinations. The Strategy pattern lets the algorithms vary independently from clients that use them.
- *Factories* – which use the Factory pattern [10] to decouple Proxy creation from Proxy use. A Factory performs the work necessary to build a Proxy, such as using a location service to find the Blob within the EMIS.

A key design goal of the Blob Streaming layer is to provide operations that behave uniformly irrespective of where the Blob actually resides or what type of Blob is being transferred. For instance, Blob Store software that receives and stores MRI images to a database remains unchanged whether the source or destination of the MRI data is in memory, on a local file, in memory of a remote client, or on disk of a remote client.

2.2 Resolving Design Challenges

Developing an enterprise-wide distributed EMIS is difficult. It requires a deep understanding of networking, databases, distributed systems, human/computer interfaces, radiological workflow, and hospital information systems. There are many technical challenges related to performance, functionality, high availability, information integrity, and security. Moreover, system requirements and the hardware/software environment change frequently.

To cope with complexity and inevitable changes, the software infrastructure of an EMIS must be flexible. In particular, developing large-scale distributed EMIS applications with low-level network programming tools like sockets is tedious, error-prone, and inflexible. Therefore, we designed Blob Streaming to elevate the level of programming for these applications. To accomplish this, we abstract away from the following tasks and mechanisms:

- *Common network programming tasks*
- *Blob location and storage mechanisms*
- *Blob type*
- *Blob transport mechanism*
- *Concurrency policies*
- *Multiple event loops*
- *Platform-specific OS mechanisms*

This section describes the software design challenges we faced when developing the Blob Streaming framework for EMIS applications. The following explains how we resolved these challenges using object-oriented design techniques, design patterns, and C++ language features. Although the discussion centers around issues that arise when building medical imaging frameworks, the principles and patterns described below are representative of a wide range of object-oriented distributed object computing environments.

```
interface BlobTransporter {
  // Timeout value representation.
  struct TimeValue { long sec; long usec; };

  // Transaction notification options.  These
  // options allow the framework to control blob
  // transfers acknowledgments.
  enum NotificationSemantics {
    SEND_NOTIFICATIONS,
    QUEUE_NOTIFICATIONS,
    IGNORE_NOTIFICATIONS
  };

  // A request to the server to send <length> bytes
  // of Blob data starting from <absoluteOffset>.
  // Since this can potentially be a long-duration
  // operation, a <timeout> can also be specified.
  // The <semantics> vary depending on the reliability
  // required.
  oneway void send (in long length,
                    in long absoluteOffset,
                    in boolean useTimeout,
                    in TimeValue timeout,
                    in NotificationSemantics semantics);

  // Informs the server to receive <length> bytes of
  // Blob data.  This data is copied to the Blob
  // starting at <absoluteOffset>.  Other options
  // are similar to send().
  oneway void recv (in long length,
                    in long absoluteOffset,
                    in boolean useTimeout,
                    in TimeValue timeout,
                    in NotificationSemantics semantics);
  // ... others omitted...
```

Figure 3: IDL Interface for Blob Transport

2.2.1 Abstracting Away from Common Network Programming Tasks

Many low-level programming tasks (such as object location and activation, parameter marshalling and framing) performed when building distributed applications are tedious and error-prone. The current version of Blob Streaming uses CORBA to automate these common low-level network programming tasks. The use of CORBA enabled us to concentrate on higher-level Blob Streaming issues (such as performance, reliability, and interface uniformity), rather than wrestling with low-level communication details. We used the following CORBA mechanisms to implement the Blob Streaming framework:

- **Strongly-typed interfaces:** In CORBA, all interfaces are defined using the CORBA interface definition language (IDL) [2]. A CORBA IDL compiler generates stubs and skeletons that translate IDL interface definitions into C++ classes. For instance, IDL interface definition in Figure 3 describes a `BlobTransporter` that is used internally by the framework to control Blob transfer from a server to a client. Client applications use the `BlobTransporter` to selectively request certain sections of a Blob. The ability to randomly access Blobs has several uses such as efficiently accessing the header information from a Blob or resuming an interrupted transaction without restarting from the beginning.

The use of CORBA IDL interfaces allows the transmission of strongly-typed data across the network. Strong typing improves abstraction and eliminates errors common to socket-level programming. For instance, if the `send` and `recv` operations shown above were implemented over a socket connection, we would need to manually convert the typed information into a stream of untyped bytes. Moreover, the sender and receiver software for parsing messages must be tightly coupled to ensure correctness. Since this provides many opportunities for errors, automating this process via CORBA significantly improves system robustness.

• **Parameter marshalling and framing:** CORBA IDL compilers automatically generate client-side stubs and server-side skeletons. These stubs and skeletons ensure correct byte ordering and linearization of all parameters sent via operation calls on CORBA interfaces over a network. For instance, the `send` and `recv` operations in the IDL `BlobTransporter` interface shown above pass various types of binary parameters. The IDL compiler maps these parameters into C++ data types such as `char` for the IDL `boolean` type and a C++ `struct` containing two `long` fields for the `TimeValue` parameter.

Marshalling the `BlobTransfer` parameters manually using sockets would require copying the parameter values into a transfer buffer and then doing a `send`. We would also have to convert the representation of the `longs` from host-byte order to network-byte order. In addition, if the bytestream-oriented TCP/IP was used, we would be responsible for framing the data correctly at the receiver. Marshalling and framing are two tedious and error-prone aspects of network programming. By using CORBA, we did not need to implement these low-level operations.

• **Object location and object activation:** CORBA supports location transparency, *i.e.*, services can be located anywhere in a distributed system. Therefore, objects accessed by clients can be remote, local (on the same host) or co-located (in the same address space). We used this feature of CORBA in the Blob Streaming framework to shield applications from the location of Blob Stores where a Blob of interest resides. Since CORBA interfaces are location independent, the framework invokes operations on Blob Stores without knowledge of where the server resides. As a result, applications that use Blob Streaming also have no dependencies on Blob Store locations.

Blob Streaming also takes advantage of CORBA's activation services. Orbix can be configured such that if a request is received for a non-active server, the a server can be launched to process the request. This allows Blob Stores to be started by Orbix only when they're needed, thus conserving system resources.

2.2.2 Abstracting Away from Blob Location and Storage

The location of Blobs can vary significantly. Blobs may exist in memory of a modality (such as an Ultrasound scanner), on the local disk of a radiologist's workstation, or in a remote Blob Store. To provide adequate reliability, availability, and performance a large-scale EMIS must support a range of Blob Stores. As shown in Figure 1, these include the following:

- *Central Blob Stores* – which provide hierarchical storage management and support long-term archiving of Blobs;
- *Cluster Blob Stores* – which cache Blobs within a cluster of diagnostic workstations in a local area network in order to increase system fault tolerance and decrease load on Central Blob Stores;
- *Workstation Blob Store* – which cache Blobs on the local disk of a diagnostic (DX) workstation;
- *Memory Blob Stores* – which cache Blobs in workstation memory.
- *New Blob Stores* (not shown) – In addition to the Blob Stores listed above, new implementations of Blob Stores can be created for more advanced data storage. For example, a *Database Store* might be designed to manage Blobs in a database (*e.g.*, Oracle, Sybase, or ObjectStore) and an *Archival Store* can be implemented to maintain legacy data to comply with legal statutes on image persistence.

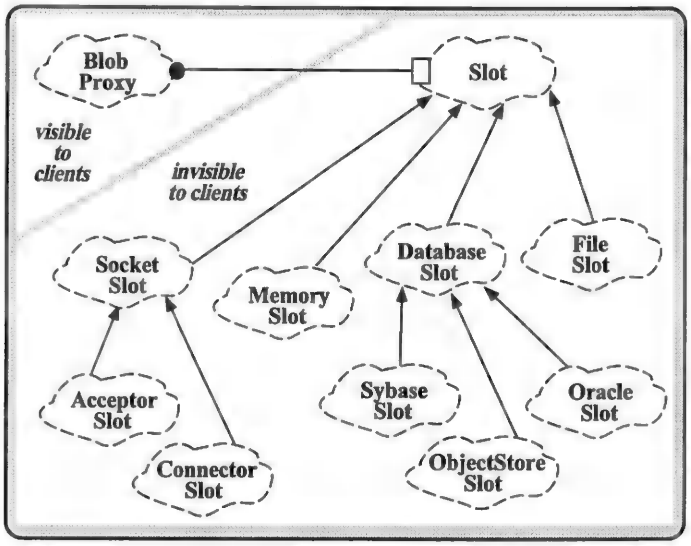


Figure 4: Blob Proxy and the Slot Hierarchy

To enhance the system usability, images must be presented to the radiologist quickly. For instance, consider the case of presenting MR images to a radiologist on a diagnostic workstation. The Blob Streaming framework is responsible for selecting the optimal transfer technique for this task. If images are stored in files on the local Workstation Blob Store, Blob Streaming memory maps the files, thereby avoiding excessive mode switches and read/write buffering. If Blobs do not reside locally, they must be found using name servers and locators[11]. Once found, they must be transported to the radiologist's workstation for display.

Before being displayed, however, Blobs may need to be processed (*e.g.*, magnified, rotated, and edge-enhanced) for optimal presentation. Due to the wide range of places that Blobs can reside, Blob Streaming allows application software that operates on Blobs to be developed independently of the Blob's location. The component in Blob Streaming that facilitates location abstraction is the *Blob Proxy*. The Blob Proxy defines the interface visible to clients. All requests to the Blob Proxy are forwarded to the Slot object, which is the abstract class that defines the interface for implementation classes. Figure 4 shows the multiple specializations of the Slot class such as socket, memory, file, and database. This is an example of the Proxy and Bridge patterns [10], where the Blob interface is decoupled from its implementation so that the two can vary independently. Section 3.1 describes the Blob Proxy programming interface in greater detail.

The Blob Streaming framework can be extended by adding new Slot implementations. The separation of interface from implementation allows these extensions to be transparent to code that uses the Blob Streaming framework. This separation also enables the Blob Streaming framework to use the same Slot implementation instance for different Blob Proxies.[3]

The advantage of defining a uniform Blob Proxy interface is to *reduce software dependencies*. Using this generic interface, application software can be written to store and retrieve images from Blob Stores, rather than to files or databases directly. This shields existing software from changes in storage type. A disadvantage to this approach is the increased learning curve. For example, developers of Blob Servers who are familiar with a particular database must learn the Blob Store interface in order to use Blob Streaming. Therefore, as discussed in Section 3.1, the Blob Streaming interface was modeled after the UNIX file system interface, which provides a uniform set of operations (like `open`, `close`, `read`, `write`, `seek`, etc.) on various types of devices, files, and I/O streams.

2.2.3 Abstracting Away from Blob Type

In addition to shielding application software from Blob location, the Blob Streaming framework abstracts away from Blob *type*. Therefore, a Blob Store that receives and stores MR images uses the same software to receive and store CT and CR images. The type of data being transferred is not directly exposed by the Blob Streaming interface.

The primary advantage of decoupling Blob type from Blob transfer is to *maximize software reuse* and *enhance interface uniformity*. In addition, our design allows meta-data (such as image identification information including patient name and examination data) to be separated and stored in a database. This decoupling allows image data (pixels) to be transported as fast as possible to the destination (*e.g.*, using memory-mapped I/O and DMA). If an application requires access to

[3]This approach is used by some CORBA implementations like Orbix where multiple proxies use the same socket channel to communicate with a server. Some slots that take relatively long to setup (such as socket slots) can be cached internally to the library and can be reused by new Blob Proxies.

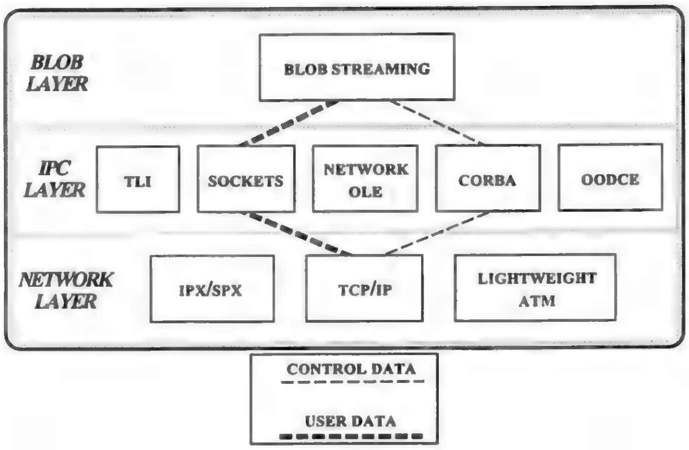


Figure 5: Communication layers used by Blob Streaming

the image's meta-data, complex queries can be performed on the database.[4]

The Blob Streaming framework is similar to the abstraction provided by an OS file system. The file system supports file format of any kind. It is up to the application using the file to correctly interpret the file format. However, the type of abstraction offered by Blob Streaming is not available in other medical imaging toolkits (such as DICOM and HL7). Many such toolkits only transfer data formatted according to the protocol's specification. This becomes a problem when trying to extend a project to deal with new data types.

Another advantage of the Blob Streaming design is that it allows the integration of image processing and Blob transfer operations. Applications need not wait for an entire Blob to transfer before processing the data (*e.g.*, compressing it as it is sent on the network and decompressing while being received). This technique is a form of Integrated Layer Processing (ILP) [12], which has been used in high-speed communication protocol stacks. ILP optimizations can improve performance significantly by overlapping communication and computation, as well as reducing memory bus traffic.

2.2.4 Abstracting Away from Blob Transport Mechanism

Blob Streaming presently uses a combination of CORBA and TCP/IP as data transport mechanisms. CORBA is used for location and control operations, whereas TCP/IP is used for bulk data transfer. This design choice reflects a tradeoff between flexibility and efficiency. Blob Streaming leverages CORBA's abstraction and flexibility, while still utilizing the efficiency of socket programming.

[4]Consistency management between pixel store and database entries are considered outside the scope of the Blob Streaming framework. Certain image formats (*e.g.*, DICOM) place meta-data as header information of the Blob. Since Blob Streaming treats all Blobs as untyped streams of data, images with integrated meta-data also can be transferred easily.

To shield applications from these low-level communication details, however, the public interface of Blob Streaming does not expose its internal transport mechanisms. This allows changes in the Blob Streaming architecture without affecting public interfaces. In particular, since CORBA is not visible to application programmers, different implementations of CORBA can be used (such as ORBeline, HP ORB Plus, or Sun NEO). Moreover, CORBA can be removed entirely and replaced with another mechanism (such as Network OLE, DCE RPC, or Sun RPC). We chose CORBA since CORBA services (such as naming and events [11] are used in other parts of our EMIS. Selecting a common distributed object computing framework reduced our training, maintenance, and software licensing costs.

Similarly, multiple transport mechanisms can be used transfer bulk data efficiently. For instance, certain types of traffic (such as video and voice) can tolerate some degree of loss. In these cases, performance can be optimized by using a lightweight ATM protocol in place of TCP/IP. Since Blob Streaming provides a layer of abstraction over these details, optimizations can be performed without altering applications.

The primary advantages of decoupling the Blob Streaming public interface from its internal transport mechanisms are to *improve flexibility*, *increase portability*, and *enable transparent performance tuning*. Therefore, the framework can be tuned to use the best performing technology without affecting applications. For instance, subsequent versions of Blob Streaming could omit TCP/IP in favor of a strictly CORBA implementation if CORBA becomes performance-competitive with lower level sockets programming. Likewise, CORBA could be replaced by Network OLE and TCP/IP replaced by a lightweight ATM protocol. Figure 5 shows the communication layers currently used by Blob Streaming. In addition, it illustrates the different IPC and network layer choices that can be used as alternatives.

One disadvantage with Blob Streaming is performance overhead of the extra level of abstraction. Although the cost of these abstractions can be reduced through optimizations such as C++ inlining, some overhead remains, as shown in Section 4. Another disadvantage is the increased complexity of the Blob Streaming internal design. In particular, connection management and synchronization are more complex. However, the complexity is not exposed to applications, which use a simple interface provided by the Blob Streaming toolkit.

2.2.5 Abstracting from Concurrency Policies

Different applications require different types of operation invocation semantics from a framework. For instance, a multi-threaded server can simplify application software by using synchronous interfaces. Conversely, a single-threaded server that cannot afford to block on a single transaction needs an asynchronous interface to all long-duration operations. Similarly, client applications are frequently single-threaded and event-driven (*e.g.*, GUIs), which cannot block indefinitely on synchronous calls.

On multi-threaded operating systems like Solaris 5.x [13] or Windows NT [14], applications can use threads to simplify programming and take advantage of parallelism. A multi-threaded application can use synchronous interfaces for long-duration operations (such as large image transfers) since it will not block other threads. In contrast, single-threaded applications must be programmed carefully to avoid starving time-critical operations by blocking on long-duration operations.

Tightly coupling an application to a particular concurrency policy increases development effort if the concurrency policy changes (*e.g.*, if a single-threaded application becomes multi-threaded or vice versa). It is hard to avoid this tight coupling because reusable frameworks and applications often must be developed without knowledge of the end system concurrency policies or hardware/software capabilities.

The Blob Streaming framework is designed to decouple application software from dependencies on concurrency policies. Blob Streaming accomplishes this by providing uniform callback-driven interfaces to both synchronous and asynchronous operations. Switching between synchronous/return-value and asynchronous/callback interfaces can require modifications to application software. For instance, consider the case where a server implemented using multiple threads is ported to a platform that does not support threads. If the software run by the threads uses synchronous interfaces, many changes will be necessary to support asynchronous transactions in a single thread.

To improve portability and uniformity, the Blob Streaming framework supports a uniform callback interface for both synchronous and asynchronous operations. These callbacks indicate when an operation completes. For instance, a single-threaded application that needs to load a large image from a remote Blob Server performs an asynchronous Blob Streaming `read`, which does not block the application from handling GUI events. When the library completes the operation, the application is notified via a callback. Similarly, synchronous Blob Streaming operations also complete with callback notifications. The difference from asynchronous calls is that the callback has already been executed when the synchronous call returns.

The advantages of abstracting away from concurrency policies are *increased uniformity* and *increased flexibility of concurrency strategies*. For instance, the same software that is used asynchronously in a single-threaded application can be used synchronously in a multi-threaded application. Because both synchronous and asynchronous operations use callbacks, switching to new concurrency policies simply requires toggling a flag. Therefore, no application software will change. This flexibility is particularly useful for developers of reusable components who write software that can be used with a variety of concurrency strategies.

The disadvantage of this approach is that some developers may never want to program asynchronous operations. To some extent, the use of uniform interfaces increases the complexity of synchronous calls in order to eliminate dependency on a particular concurrency model. To address this issue, the Blob Streaming library offers wrappers around the synchronous callback operations to provide a synchronous/return-value API. This is illustrated in Section 3.4.

2.2.6 Abstracting Away from Event Loops

Complex EMIS applications must react to events from multiple sources. Common sources of EMIS events include DICOM toolkits, HL7 interface engines, GUI window events, and Blob Streaming transfers. Furthermore, the Blob Streaming library must integrate the processing of socket-level events, CORBA events, timer events, and signals.

Each of these sources of events (X Windows, CORBA, etc.) has its own event loop. If an application must react to all of these events, it can not block indefinitely on any one event loop. One solution is to use a polling technique where the application uses a round-robin policy to check each event loop. A disadvantage to this approach is that it can lead to excessive overhead when there are no pending events.

An alternative approach is to combine the multiple event loops into a single waitable object. Blob Streaming uses ACE's Reactor[9] to implement this technique. The Reactor provides a mechanism that integrates the event demultiplexing and event handler dispatching components of multiple frameworks. It presents applications with an object-oriented interface to lower-level OS event demultiplexing mechanisms that react to I/O handle events, timer events, and signal events. The `select`, `poll`, and `WaitForMultipleObjects` system calls are common examples of these demultiplexing mechanisms. This allows the application to block on the Reactor for all events, eliminating the overhead imposed by the polling technique.

Frameworks such as X windows or CORBA are generally driven by events from "waitable" I/O handles (also called descriptors). We will use a UNIX-centric naming policy and call these *select-based* objects. Some applications and frameworks also use waitable resources such as message queues, semaphores, and condition variables. We will call these *synchronization-based* objects.

An example of a synchronization-based object exists in MT-Orbix. The MT-Orbix library dedicates a thread to each network connection. This allows easy integration with third-party toolkits (such as the Tuxedo transaction monitor) that utilize System V message queues (which are synchronization-based). Requests that come over the connections are queued up in a thread-safe message queue. The main thread of control now waits on a conditional variable, rather than waiting in a demultiplexing operation (like `select` or `poll`) as it would in the single-threaded. After a new request is added to the message queue, the main thread is signaled, which then dequeues and processes the request.

The MT Orbix model adds a synchronization-based source of event demultiplexing to applications. For Win32 platforms, `WaitForMultipleObjects` can be used to wait on both select-based and synchronization-based objects. However, this is problematic for platforms (such as SVR4 UNIX) that do not provide a uniform model for demultiplexing synchronization-based events. The problem is relatively easy to solve if applications can use multiple threads. In this case, one or more threads could be dedicated to process select-based events, while other threads could be dedicated to process the synchronization-based events queued in the message queue.

However, many systems (including ours) have to deal with large amounts of legacy code that is non-reentrant. Therefore, it becomes essential that the select- and synchronization-based events be combined into one logical source. The following components are used to adapt the MT-Orbix I/O handles into a single demultiplexing object:

- *Select-based event handler* – The main thread is dedicated to handling the select-based events. This is done through the `select` demultiplexing operation.
- *Synchronization-based event handler* – A separate thread is dedicated to handling the synchronization-based MT-Orbix events. This is done with MT-Orbix's `impl_is_ready` and operation marshalling filters. Orbix filters allow applications to access incoming CORBA requests from the message queue before they are invoked on the appropriate objects.
- *Select-synchronization bridge* – An intra-process communication channel (such as a `pipe` or local `socket`) is created for communication between the two event handlers. The reading end of the `pipe` is owned by the main thread and is registered with the demultiplexing operation for input events. The secondary thread owns the writing end of the `pipe`. Orbix events are dequeued by the secondary thread from the message queue and forwarded to the main thread through the `pipe`. The main thread reads the event from the pipe handle and processes the event.

This design restores the simple model of "single threaded, single source of events" that our legacy applications required.

The advantage of integrating multiple event loops is that it allows developers to use Blob Streaming *while continuing to integrate with other frameworks*. For instance, an application developer building X-window applications can perform Blob Streaming operations without changing how the application interfaces with the event-loop. Since Blob Streaming uses the Reactor, the framework can be integrated with the necessary event-loop without affecting internal framework software or external framework interfaces.

The disadvantage to this approach is that the Reactor must be integrated with each new framework. This integration can be difficult if the framework does not provide adequate hooks into its internal event demultiplexing logic. Moreover, there is a performance penalty for this integration. For instance, the approach we used to integrate MT Orbix with our single-threaded applications effectively eliminated concurrency within the event demultiplexing layer of imaging applications.

2.2.7 Abstracting Away from Platform-specific OS Mechanisms

As shown in Figure 2, the Blob Streaming framework shields applications from non-portable OS-specific features such as memory mapping, event demultiplexing, multi-threading, and interprocess communication. This, in turn, makes applications using the Blob Streaming interface portable across platforms *without* changing application communication software. The Blob Streaming framework has been ported to a variety of UNIX platforms, as well as Win32 platforms [14].

The primary advantage of decoupling application software from OS-specific mechanisms is *cross-platform portability*. The primary disadvantage is that performance and functionality may be compromised to provide a generic OS interface. For example, the current version of Blob Streaming does not yet take advantage of native Windows NT asynchronous I/O mechanisms such as overlapped I/O or I/O completion ports [15].

3 Blob Streaming Interfaces and Examples

This section describes the key components in the Blob Streaming framework and illustrates how to use these components to program synchronous and asynchronous Blob transfer applications. Our goal is to demonstrate the expressive power and simplicity of the framework.

3.1 Blob Proxy

Figure 6 shows the interface of the `Blob Proxy` class, which includes methods like `open`, `close`, `read`, `write`, `size`, and `position`. These methods are similar to those provided by System V Release 4 (SVR4) UNIX for file I/O. SVR4 UNIX adapts a wide variety of disk and communication devices into a common set of I/O operations. Blob Streaming has the following notable differences from the SVR4 UNIX file system interfaces, however:

- *Seamless Integration of Memory, Networking, and File I/O* – The SVR4 UNIX I/O interfaces are not entirely uniform. For instance, a different set of calls is required to open a socket vs. opening a file. Likewise, SVR4 UNIX uses a different interface for memory-mapped file I/O and buffer-based network/file I/O. In contrast, Blob Streaming provides a uniform interface for all these forms of I/O. This makes it possible to abstract away from Blob location by removing inconsistencies and special cases in the I/O programming model.
- *Object-oriented interfaces* – Low-level network programming tools such as sockets do not provide sufficient type-checking since they utilize untyped I/O handles. It is disturbingly easy to misuse these interfaces in ways which can only be detected at run-time (such as trying to read or write data on a passive-mode listener socket used to accept connections). Unlike SVR4 UNIX, which provides these C-level system call interfaces, Blob Streaming provides C++ interfaces. The use of C++ enforces encapsulation and yields a more modular, extensible, and less error-prone programming interface, without compromising performance.

The `BlobProxy` interface is designed so that operations can be invoked synchronously or asynchronously. Asynchronous invocation is useful for long-duration operations (such as `open`, `send`, and `recv`) that can run independently without blocking the main thread of control. Synchronous operations are useful for (1) short-duration operations (such as `size` and `type`) that do not block the caller for long and (2) applications that spawn multiple threads to execute the calls without blocking the entire process.

The `SynchOptions` class gives users a single interface to specify the type of synchrony/asynchrony policy to be used for a call. This encapsulation simplifies the Blob Streaming interfaces and gives applications greater flexibility over the synchronization policies used by the application. For instance, applications can define a global instance of `SynchOptions` which is passed in to every Blob Streaming operation. In this way, applications can change the synchronization policy used by the entire application through a single `SynchOptions` instance. The `SynchOptions` interface is defined as follows:

```
class SynchOptions
{
  // Options flags for controlling synchronization.
  enum Options {
    NONBLOCK, // Use asynchronous invocation.
    BLOCK, // Use synchronous invocation.
    TIMEOUT // Use timed invocation.
  };

  SynchOptions
    (Options options, // Synch policy.
     const TimeValue &timeout, // Timeout duration.
     LocalReceiver *notifiee = 0); // Who to notify.

  // ...others omitted...
}
```

The `Options` enumeration records whether the call is to be made synchronously or asynchronously and whether is should be timed or not. If the `TIMEOUT` enumeral is enabled, then the `TimeValue` is used to specify the timeout duration. Finally, if the call is performed asynchronously, the `LocalReceiver` pointer is used to specify an object who `receiveNotification` method is called back when the asynchronous invocation completes.

3.2 Blob Proxy Factory

The Blob Proxy Factory is responsible for creating proxies to Blobs that may be remote or local. The Factory is also responsible for dynamically selecting and configuring the objects (such as Slots) needed to implement the Blob Proxy interface. This encapsulation of the responsibility and

```
class BlobProxy
{
public:
  // Open the Blob Proxy.
  void open (const SynchOptions &options);

  // Close the proxy down and release resources.
  void close (void);

  // Read <numBytes> from the Blob Proxy into the
  // <buffer>.
  void read (Buffer &buffer,
             size_t numBytes,
             const SynchOptions& options);

  // Write <numBytes> from the <buffer> to the
  // Blob Proxy.
  void write (const Buffer &buffer,
              size_t numBytes,
              const SynchOptions& options);

  // Size of data represented by the Blob Proxy.
  size_t size
    (const SynchOptions& options) const;

  // Type of data represented by the Blob Proxy.
  // Various types include pixel data or DICOM
  // image.
  BlobProxy::Type type
    (const SynchOptions& options) const;

  // Set/Get the position of the Blob Proxy
  // This allows the user to move to a
  // particular location in the Blob.
  void position (size_t offset,
                 BlobProxy::OffsetSetting whence,
                 const SynchOptions& options);
  size_t position
    (const SynchOptions& options) const;

  // ...others omitted...

private:

  // A Blob Proxy can only be created
  // by a Blob Proxy Factory.
  BlobProxy (const BlobKey &key);
};
```

Figure 6: Blob Proxy Interface

```
class BlobProxyFactory
{
public:

  // The factory creates a new Blob Proxy that is
  // bound to an existing Blob represented by the <key>.
  static
  BlobProxy *bindBlob (const BlobKey &key,
                       const SynchOptions &options);

  // The factory creates a new Blob (represented by
  // <key>) of <size> bytes. It also creates a Blob
  // Proxy that is bound to the new Blob.
  static
  BlobProxy* routeBlob (const KBlobKey &key,
                        size_t size,
                        const SynchOptions &options);

  // ... others omitted...
};
```

Figure 7: Blob Proxy Factory Interface

process of creating and composing implementation objects for the Blob Proxy isolates the user of the proxies from the implementation classes.

Figure 7 shows the interface of the `Blob Proxy Factory` class, which has methods like `bind` and `route`. The `bind` method creates a Blob Proxy that is bound to a Blob. This is similar to the functionality provided by CORBA for creating a proxy to a remote object. The `route` method is used to create a new Blob of a given size. In this case, the factory is responsible for communicating with the appropriate Blob Store to reserve space for the new Blob. If the space is successfully reserved, a proxy is created to the new Blob and returned to the user.

3.3 Blob Transporters

The Blob Transporter is responsible for optimally copying data from one Blob to another. The Blob Transporter implements algorithms that iterate over the source Blob and copy the data to the destination Blob. The `copy` methods of the Blob Transporter are similar to the algorithms provided by the C++ Standard Template Library (STL) [16]. The STL algorithms are completely generic and behave the same way irrespective of the types they work on. In contrast, the algorithms defined by the Transporter are optimized for different Blob locations. Since there are relatively few Blob locations types (memory, file, network, and database), it pays to explicitly optimize the Blob Transporter for each. For instance, a transporter can simply before a `memcpy` when the source and destination of a copy are both memory.

Figure 8 shows the interface of the `Blob Transporter` class. Note that the `CopyTransporter` only implements static interfaces. State for copies in-progress is dynamically allocated by the copy routine and deleted when the operation completes. If the state for a copy operation was kept as instance data of a `CopyTransporter` instance, the instance would only be able to keep track of one in-progress copy.

```
class CopyTransporter
{
public:

  // Copy entire <source> Blob to
  // <destination> Blob.
  static
  void copy (BlobProxy *destinationProxy,
             BlobProxy *sourceProxy,
             const SynchOptions &options);

  // Copy <size> bytes from <source> Blob
  // to <destination> Blob.
  static
  void copy (BlobProxy *destinationProxy,
             BlobProxy *sourceProxy,
             size_t size,
             const SynchOptions &options);

  // ... others omitted...
};
```

Figure 8: Blob Transporter Interface

This would also force the user to create and manage multiple instances of CopyTransporter in order to execute multiple copy operations simultaneously.

3.4 Using the Blob Streaming Framework

The following discussion presents several use-cases that illustrate how to program synchronous and asynchronous applications using Blob Proxies. The two examples in Figures 9 and 10 use Blob Streaming to copy images from a remote Blob Store to a local Blob Store. Blobs in the system are identified uniquely by BlobKeys. Both examples copy an image identified by sourceKey to an image identified by destinationKey.

A destinationKey is created by replicating the sourceKey and changing the host information in the destinationKey to the local host. Space is then reserved for the new image at the local BlobStore by calling BlobStreamingFactory::routeBlob. The copy options sets a timeout of 30 seconds for the copy operation. The copy operation will timeout if the operation does not complete in the specified time. In the synchronous example, an exception is raised in the event of failure or timeout. In the asynchronous example, the Replicator class is notified of the result of the operation. Exceptions cannot be raised in the asynchronous example since the call to the copy method returns immediately without blocking the caller.

The primary difference between the two examples is the nature of the copy call. The first example shown in Figure 9 uses a synchronous, return-value based version of the CopyTransporter::copy method call. The second example shown in Figure 10 uses the asynchronous, callback based version of the CopyTransporter::copy method call.

Figure 11 illustrates the method that receives copy notifications for callback-based copies. The following changes to

```
// Retrieve to local store.
void copy (BlobKey sourceKey ) {

  // Create a key for the destination
  BlobKey destinationKey (sourceKey,
                          localHostName);

  // Allocate space on Blob Store for
  // destination Blob.
  BlobStreamFactory::routeBlob (destinationKey,
                                source->size ());

  // timeout after 30 seconds
  TimeValue timeout (30);
  SynchOptions copyOptions
    // Synchronous, timed invocation.
    (SynchOptions::TIMEOUT |
     SynchOptions::BLOCK,
     timeout); // Amount of time to block.

  // Synchronous copy of the Blob.
  try {
    CopyTransporter::copy (sourceKey,
                           destinationKey,
                           copyOptions);
  } catch (RecoverableException exc) {
    switch (exc.tag ()) {
    case ERROR_BLOB_COPY_FAILED:
      // report failure
      break;
    case ERROR_BLOB_COPY_TIMEOUT:
      // report timeout
      break;
    }
  }
  // report success
}
```

Figure 9: Synchronous, Return-value-based Copy Example

```
class Replicator
  : public LocalReceiver
    // Defines the pure virtual
    // receiveNotification() method.
{
public:
  // Handles I/O completion.
  virtual bool receiveNotification
    (LocalNotification *notification);

  // Retrieve to local store.
  void copy (BlobKey sourceKey) {
    // Create a key for the destination
    BlobKey destinationKey (sourceKey,
                            localHostName);

    // Allocate space on Blob Store for
    // destination Blob.
    BlobStreamFactory::routeBlob (destinationKey,
                                  source->size ());

    // Timeout after 30 seconds.
    TimeValue timeout (30);
    SynchOptions copyOptions
      // Asynchronous, timed invocation.
      (SynchOptions::TIMEOUT |
       SynchOptions::NONBLOCK,
       timeout, // Amount of time to wait.
       this); // Notify this object (Replicator)
              // upon completion of the copy.

    // Start an asynchronous copy.  On completion,
    // our receiveNotification() method is called.
    CopyTransporter::copy
      (sourceKey, // copy from this Blob
       destinationKey, // to this Blob
       copyOptions); // copy options
};
```

Figure 10: Asynchronous callback-based copy example

```
bool Replicator::receiveNotification
  (LocalNotification *notification)
{
  CopyTransporter::CopyNotification *
    copyNotification =
    (CopyTransporter::CopyNotification *)
    notification;

  switch (copyNotification->result ()) {
  case CopyTransporter::CopyNotification::SUCCEEDED:
    // report failure
    break;
  case CopyTransporter::CopyNotification::FAILED:
    // report failure
    break;
  case CopyTransporter::CopyNotification::TIMEOUT:
    // report timeout
    break;
  default:
    return 0;
  }
};
```

Figure 11: Receiver of Copy Notifications

Replicator::copy are all that are required to make an asynchronous, callback-based operation like this:

```
SynchOptions copyOptions
  // Asynchronous, timed invocation.
  (SynchOptions::TIMEOUT |
   SynchOptions::NONBLOCK,
   timeout, // Amount of time to wait.
   this); // Notify this object upon
          // completion of the copy.
```

into a synchronous, callback-based operation like this:

```
SynchOptions copyOptions
  (SynchOptions::TIMEOUT, timeout);
```

If the thread executing the copy operation can afford to block without compromising the quality of service of other components of the application, the synchronous approach can be used. However, if the long-duration copy operation will affect other components of the application, the asynchronous approach can be used. This allows the application developer to develop the imaging replication module without becoming dependent on the concurrency model used for image replication. As a result, systems that use Blob Streaming are more portable than those written to use lower-level OS mechanisms directly.

4 Performance of the Blob Streaming Framework

Sections 2.2 and 3 motivate and outline the design and use of the Blob Streaming framework. This design abstracts away from many low-level communication tasks to achieve the flexibility requirements of distributed EMISs. In practice, however, we recognized that the framework will not be widely used unless applications built using it meet their performance requirements. This section describes performance tests of the Blob Streaming framework. The test scenario involved the point-to-point transfer of Blobs between a client and a server. In a large-scale EMIS, several types of bulk data transfers can place high loads on a communications framework. For instance, transferring a typical MR image study can include fifty 250 Kbyte images. Likewise, a CR image study can include several 500 Kbyte images. The tests performed on the Blob Streaming framework have been designed to mimic the behavior of transmitting studies such as these.

4.1 Test platform and benchmarks

The performance results in this section were collected using a Bay Networks LattisCell 10114 ATM switch connected to two dual-processor SPARCstation 20 Model 712s running SunOS 5.4. The LattisCell 10114 is a 16 Port, OC3 155Mbs/port switch. Each SPARCstation 20 contains two 70 Mhz Super SPARC CPUs with a 1 Megabyte cache per-CPU. The SunOS 5.4 TCP/IP protocol stack is implemented

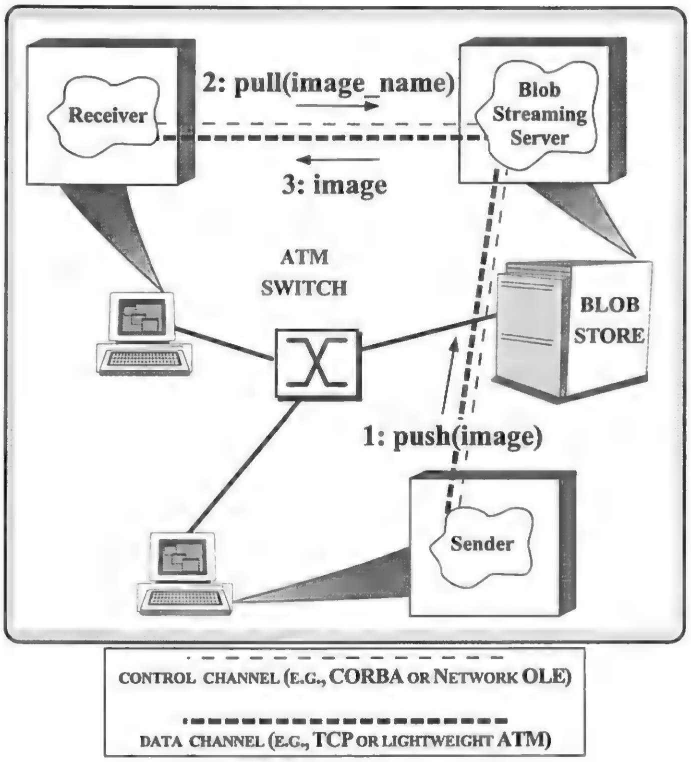


Figure 12: Push and Pull Models

using an optimized version of the STREAMS communication framework [17]. Each SPARCstation has 128 Mbytes of RAM and an ENI-155s-MF ATM adaptor card, which supports 155 Megabits per-sec (Mbps) SONET multimode fiber. The Maximum Transmission Unit (MTU) on the ENI ATM adaptor is 9,180 bytes. Each ENI card has 512 Kbytes of on-board memory. A maximum of 32 Kbytes is allotted per ATM virtual circuit connection for receiving and transmitting frames (for a total of 64 K). This allows up to eight switched virtual connections per card.

Data for the experiment was produced and consumed by a client and server test application. The client represents a diagnostic workstation. The server application represents a Blob Store server. Various client and server parameters may be selected at run-time. These parameters include the size of the Blob being transferred and the size of the socket transmit and receive queues.

Our test environment is similar to the widely available `ttcp` benchmarking tool. However, our test application differs from `ttcp` since we implement a "request/response" model rather than the conventional `ttcp` "flooding" model. In our model, the client can request the server to send it data (the "pull" model) or move data to the server (the "push" model). This is different from `ttcp` because the data transmitter does not simply flood the receiver with a continuous unidirectional stream of bytes. The push and pull models implemented by our test application are illustrated in Figure 12 and described below.

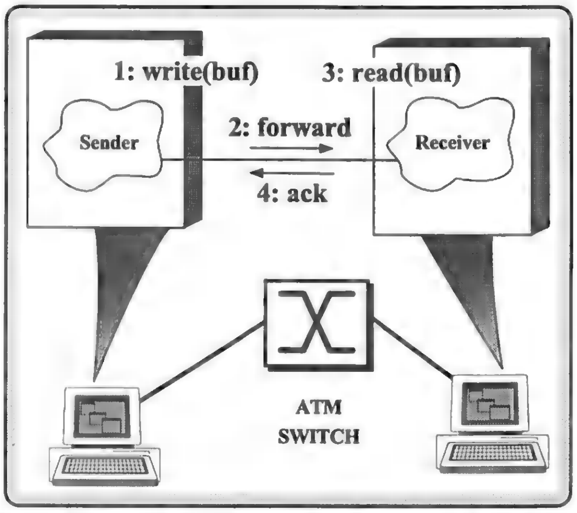


Figure 13: C and C++ `ttcp` Benchmarking Architecture

• **The push model:** This model is representative of the use case where a modality stores data on a Blob Store. In addition, it can be used by a Blob Store to precache data to a workstation. The push model behaves as follows:

1. *Negotiation* – the client sends control data to the server characterizing the image being transferred from the client to the server (*e.g.*, size and name of the image);
2. *Transmission* – the client then sends the image data;
3. *Confirmation* – the server sends a confirmation to the client when all the data is received. This acknowledgment is necessary to insure end-to-end reliability of the request/response transaction.

• **The pull model:** This model is representative of the use case where a workstation retrieves data from a Blob Store. The pull model behaves as follows:

1. *Negotiation* – the client sends control data to the server characterizing the image the client wants from the server (size and name of the image)
2. *Transmission* – the server then sends the image data. Once the client receives the data that was requested from the server, the request/response transaction is complete. Note that the pull model does not require an extra acknowledgment, which improves performance.

We implemented and benchmarked the following versions of the test application for Blob transfers:

• **C version:** this is implemented completely in C. It uses C socket calls to transfer and receive the data and control messages via TCP/IP. Figure 13 illustrates the design of this `ttcp` test.

• **ACE C++ version:** this version replaces all C socket calls in the applications with the C++ wrappers for sockets provided by the ACE network programming components [6].

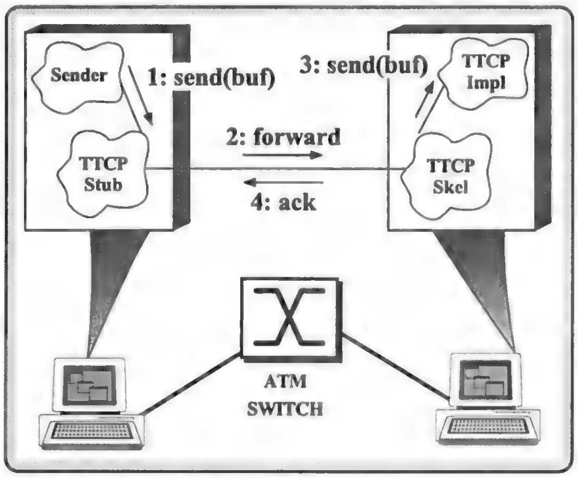


Figure 14: CORBA `ttcp` Benchmarking Architecture

ACE encapsulates sockets with typesafe, portable, and efficient C++ interfaces. Figure 13 illustrates the design of this test, as well.

■ **CORBA version:** the Orbix implementation of CORBA was used: version 1.3 of Orbix from IONA Technologies. This version replaces all socket calls in the test applications with stubs and skeletons generated from a pair of CORBA interface definition language (IDL) specifications. One IDL specification uses a `sequence` parameter for the data buffer and the other uses a `string` parameter. Figure 14 illustrates the design of this test.

■ **Blob Streaming version:** the Orbix implementation of CORBA was used to exchange control messages and C++ wrappers for sockets provided by ACE were used for bulk data transfer. This is the only test that implements both the push and pull models. Figure 15 illustrates the design of this test for the push model and Figure 16 illustrates the design of the test for the pull model.

4.2 Performance Results

4.2.1 Throughput Results

We ran a series of tests that transferred 1 MB, 8 MB, 16 MB, and 32 MB of user data using TCP/IP over our ATM network testbed. Two different sizes for socket queues were used: 8 K (the default on SunOS 5.4) and 64 K (the maximum size supported by SunOS 5.4). Each test was run 20 times to account for performance variation due to transient load on the networks and hosts. The variance between runs was very low since the tests were conducted on an otherwise idle network.

• **Push Model Throughput:** Figure 17 shows that different versions of tests for Ethernet show much less variation, with the performance for all tests ranging from around 8 to 8.7 Mbps with 64 K socket queues. Figure 17 also summarizes the performance results for all the push model benchmarks

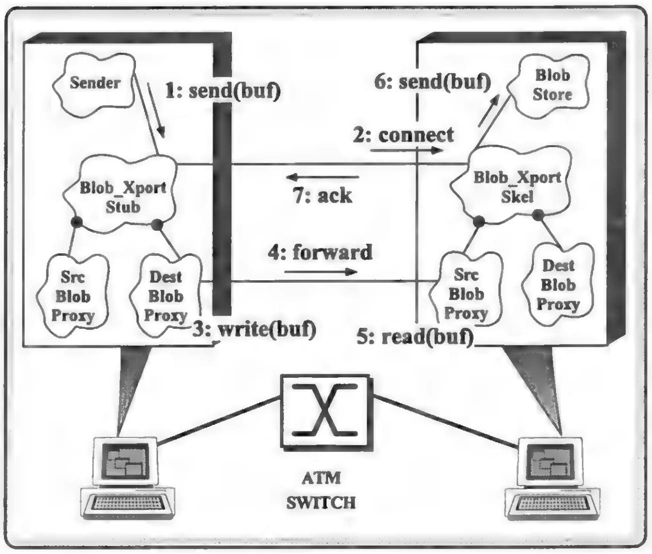


Figure 15: Blob Streaming `ttcp` Benchmarking Architecture (Push Model)

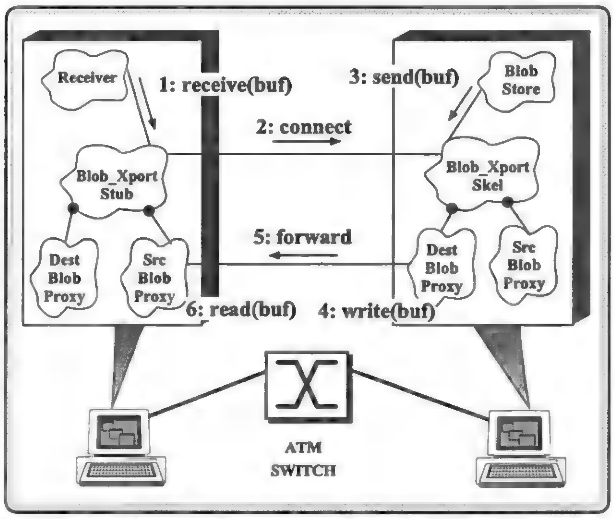


Figure 16: Blob Streaming `ttcp` Benchmarking Architecture (Pull Model)

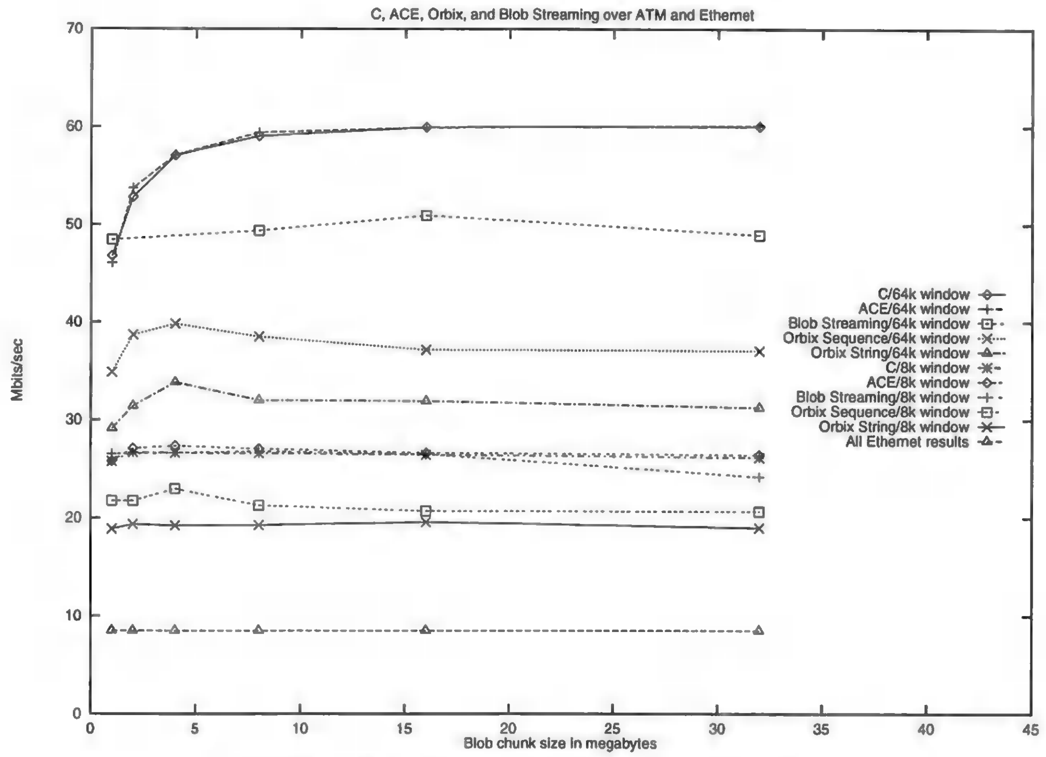


Figure 17: Push Model Performance Over Ethernet and ATM

using 64 K and 8 K socket queues over a 155 Mbps ATM link.

The following describes the performance of each test program, using 64 K and 8 K socket queues:

- The C and ACE C++ wrapper versions of the tests obtained the highest throughput: 60 Mbps using 64 K socket queue. This indicates that the performance penalty for using the higher-level ACE C++ wrappers is insignificant and is comparable with using low-level C socket library calls directly.
- The Blob Streaming performance was slightly more than 80% of the C and C++ versions, reaching 50 Mbps with 64 K socket queues. The primary source of overhead in the Blob Streaming framework is explained in Section 4.2.2.
- The Orbix `sequence` version peaked at around 66% of the C and C++ versions, reaching 40 Mbps, whereas the Orbix `string` implementation peaked at 33 Mbps (both using 64 K socket queues). The primary sources of overhead for the Orbix implementation of CORBA is explained in Section 4.2.2.

In addition to comparing the performance of the various transport mechanisms, Figure 17 also illustrates the generally low level of utilization of the ATM network. In particular, 60 Mbps represents only 40% of the 155 Mbps ATM link. This disparity between network channel speed and end-to-end application throughput is known as the *throughput preservation problem* [18]. This problem occurs when only a portion of the available bandwidth is actually delivered to applications.

The throughput preservation problem stems from operating system and protocol processing overhead (such as data movement, context switching, and synchronization [12]). This throughput preservation problem is exacerbated by contemporary implementations of distributed object computing middleware like CORBA, which copy data multiple times during fragmentation/reassembly, marshalling, and demarshalling. Furthermore, the latency associated with the request-response protocol implemented by `ttcp` significantly reduced performance. An earlier implementation of `ttcp` [3] attained 90 Mbps over the same ATM testbed by using a "flooding" traffic generation model that did not use an end-to-end acknowledgment scheme.

Finally, Figure 17 illustrates the impact of socket queue size on throughput. Increasing the socket queue from 8 K to 64 K doubled performance from 28 Mbps to 60 Mbps. The reason for this is that larger socket queues increase the TCP window size [19], which allows the transmission of multiple TCP segments back-to-back.

These socket queue results demonstrate the importance of having hooks to manipulate underlying OS mechanisms (such as transport layer and socket layer options). It is important to note that the choice of socket queue size has more impact than the choice of communication model (*i.e.*, C/C++ vs. CORBA vs. Blob Streaming). In fact, the slowest communication model (CORBA) is faster with 64 K socket queues than the faster communication model (C/C++) with 8

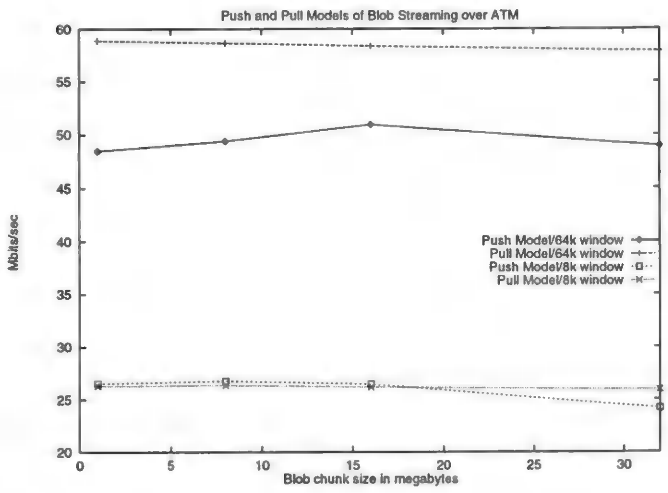


Figure 18: Pull Model vs. Push Model Performance Over ATM for Blob Streaming

K queues. Clearly, communication frameworks that do not offer these hooks to application developers are destined to perform poorly over high-speed networks.

• **Pull Model Throughput:** Figure 18 compares the performance of the pull model and the push model of the Blob Streaming versions of the tests.[5] For 64 K socket queue size, the pull model out-performed the push model by 15% to 20% for all sizes of data being transferred. This result illustrates the drawback of the push model, which must wait for an acknowledgment from the receiver in order to guarantee end-to-end delivery.

Figure 18 also compares the two models with 8 K socket queue sizes. There is no appreciable difference in performance of the two models with this socket queue size. This illustrates once again how important it is for ORBs to allow applications to tune the size of the underlying socket queues.

4.2.2 High Cost Functions

In order to explain the throughput results shown above, we used the `Quantify` execution profiler [20] to pinpoint the sources of overhead. The test applications were relinked using `Quantify`, which modified the object code to include monitoring instructions. Two related tools (`qxprof` and `qv`) were then used to display and measure the amount of time spent in functions during program execution. Figure 19 lists the functions where the most time was spent sending and receiving 1 Mbytes of user data and using 64 K socket queues. The results show the push model experiment repeated 100 times.

• **High cost operations for C and ACE C++:** The high cost operations for C and ACE C++ wrapper versions are almost identical. The sender spent 94% of the time in the

[5]Due to space constraints, the ACE, C, and CORBA pull model results are not shown – they exhibit similar performance curves, however.

Test	%Time	#Calls	Name
C sockets	93.9	112	`write`
(sender)	3.6	110	`read`
C sockets	93.2	13,085	`read`
(receiver)	4.5	102	`write`
ACE C++ wrapper	94.4	112	`write`
(sender)	3.2	110	`read`
ACE C++ wrapper	93.9	12,984	`read`
(receiver)	5.6	102	`write`
Orbix Sequence	53.5	127	`write`
(sender)	35.1	223	`read`
	7.3	1,108	`memcpy`
Orbix Sequence	84.6	12,846	`read`
(receiver)	12.4	1,064	`memcpy`
	3.2	101	`write`
Orbix String	45.0	127	`write`
(sender)	35.1	223	`read`
	10.8	1,315	`strlen`
	6.0	1,108	`memcpy`
Orbix String	70.7	12,443	`read`
(receiver)	16.1	2,142	`strlen`
	10.0	1,064	`memcpy`
	3.0	101	`write`
Blob Streaming	48.8	327	`write`
(sender)	44.8	232	`read`
	1.3	2,055	`memcpy`
Blob Streaming	77.2	12,546	`read`
(receiver)	16.4	12,734	`memcpy`
	1.4	102	`write`

Figure 19: High cost Functions for `Push` Model Blob Streaming Tests

`write` system call sending data to the receiver. About 3% of the time was spent in receiving acks from the receiver. The receiver spent 93% of the time in the `read` system call receiving data from the sender. About 1.5% of the time was spent in sending acks from the sender.

The sender approximately made 100 `write` system calls (once per iteration) to send the data and approximately 100 `read` system calls (once per iteration) to receive the ack. The receiver made approximately 13,000 `read` system calls (130 times per iteration) to receive the data and approximately 100 `write` system calls (once per iteration) to send the ack. The excessive amounts of `reads` results from fragmentation of the data into packets of 9,180 bytes, which is the maximum transmission unit (MTU) size of the ATM network.

• **High cost operations for Orbix:** Two different implementations of Orbix were profiled. The first version uses a `sequence` parameter for the data buffer and the other uses a `string` parameter. Both the sender and the receiver spent a considerable amount of time in copying data (6-12% of the time was spent in `memcpy`), slowing down the performance of the system. The decrease in performance compared to the C and ACE wrappers versions causes the sender to wait longer to receive the ack from the receiver. This is indicated by the time spent in the `read` system call. These experiments are similar to the ones in [3] and details about the behavior is explained in that paper.

The Orbix implementation differs from the C and ACE implementations in the number of `read` system calls made to receive an ack. Orbix implementations make two `read` system calls per ack compared to one call by the C and ACE versions. This is because Orbix uses the "header followed by the data" protocol. The first `read` system call reads the fixed size header and the subsequent `read` system call reads the variable size payload. This protocol is not necessary in the C and ACE versions as the only type of information sent to the sender is an ack.

Figure 17 illustrated that the performance of the Orbix `sequence` results consistently performed around 6 to 7 Mbps higher than the `string`. This difference in performance is due to the C++ mapping for strings in the CORBA IDL specification. The client-side stubs that perform parameter marshalling for remote calls must obtain the length of the string being sent. This is accomplished via calls to `strlen`, which add significant overhead to the `string` version. However, the IDL-to-C++ mapping of the `sequence` provides length fields in addition to the data.

To illustrate the difference, consider the following IDL definition of a `sequence` and it's corresponding C++ mapping:

```
// IDL definition
typedef sequence<char> char_sequence;
oneway void push (in char_sequence data_seq,
                  in string data_string);

// C++ mapping
struct char_sequence {
  u_long _maximum;
  u_long _length;
  char *_buffer;
};

void push (const char_sequence &data_seq,
           const char *data_string);
```

The `_length` field is explicitly set by the application allowing client-side stub to know the size of the `_buffer`. Thus, `data_string` requires a `strlen`; `data_seq` does not.

• **High cost operations for Blob Streaming:** Compared with the C, ACE, and Orbix implementations, the Blob Streaming sender implementation performs a higher number of `write` calls. As shown in Figure 19, Blob Streaming makes three `write` system calls per iteration while the C, ACE, and Orbix versions make one call. The first call by Blob Streaming sends the control information, the second call is for the data, and the third is for a request for the ack. The control information cannot be bundled with the data as Blob Streaming uses different channels for control and data messages. All the other versions use the same channel for control and data messages.

The `Quantify` analysis of the Blob Streaming implementation revealed that the receiver spent 16.4% of the time in `memcpy`. Upon closer inspection, we found our implementation was making an extra copy of the data received from clients. A single extra copy reduced the performance of Blob Streaming and the sender has to wait longer to receive an ack from the receiver.

One way to reduce this overhead is to have the application preallocate the buffer space before passing into the Blob Stream receiver. Once we remove the extra data copy from the receiver, we expect the results to perform roughly the same as the C and ACE C++ wrapper versions. In particular, although the sender makes three times more calls to `write`, we expect the overhead is due to the extra data copying on the receiver, rather than the additional mode switching on the sender.

5 Evaluations and Recommendations

When developing large frameworks such as Blob Streaming, the greatest challenge is designing for the future. The framework must be able to adapt to the ever-changing needs of the customer it is built for. Blob Streaming chose CORBA as a tool to help the framework meet these demands. The following two sections discuss our recommendations to others facing similar challenges.

5.1 Designing Object-Oriented Communication Frameworks

Based on our performance experiments and our experience using the Blob Streaming framework, our evaluations and recommendations for developing object-oriented communication frameworks for high-performance bulk data delivery systems include the following:

- *Flexible tools* – The framework must be able to deal with new types of data and new transport protocols and networks. If the tools used to build the framework are not flexible to changing needs, the framework will not be flexible either. This was one of our motivations for CORBA.
- *Performance* – Meeting the performance requirements of bandwidth-intensive and delay-sensitive applications is essential before the framework will be adopted widely. This motivates our combination of CORBA with lower-level transport mechanisms in order to gain the performance benefits of sockets.
- *Ease of use* – The learning curve of using a new framework must be as small as possible. This force inspired us to simplify the Blob Streaming interfaces by modeling after the UNIX file I/O interfaces and including abstractions such as `SynchOptions` and the stateless `CopyTransporter`.
- *Concurrency Policy* – The framework should try to avoid making concurrency policy decisions. The framework must, however, provide mechanisms that allow the framework to work correctly in a multi-threaded / multi-process environment. Blob Streaming addresses this need by supporting uniform callback interfaces for both synchronous and asynchronous operations.

- *Portability* – Portability requirements of the framework must be addressed in the early phase of design. This helps the designers and developers make reasonable assumptions about the OS level services available. Blob Streaming uses a toolkit of C++ wrappers [6] to remove dependencies from OS-specific system call mechanisms.
- *Transport Protocols and Networks* – Networks have experienced a tremendous growth in the last few years. There is no reason to doubt that this trend will continue for many more years. Prototypes of gigabit network are already being developed [21]. Next generation frameworks should be able to adapt to new networks and new transport protocols.
- *Event-Loop ownership* – The framework should not assume ownership of the event-loop. Applications using the framework will typically be dealing with multiple sources of input like GUI events and CORBA events. Blob Streaming uses the ACE Reactor[9] as a single demultiplexing object to encapsulate these multiple sources of events.

5.2 Using CORBA Effectively

CORBA offers many advantages for developing complex distributed systems since it automates many common network programming tasks such as object selection, location, and activation, as well as parameter marshalling and framing. However, a major disadvantage of CORBA is that current implementations incur significant performance overhead when used to transfer large amounts of data [3].

We addressed the performance problems of CORBA by integrating it with sockets. Our approach uses CORBA for control messages and sockets for bulk data transfer. This two-tiered design leverages CORBA's extensibility and socket's efficiency. CORBA is particularly useful for short-duration, request/response operations that exchange richly typed data. Modifying or extending the type of information exchanged between applications is also straightforward since CORBA automatically generates code to marshall the parameters. Thus, for many types of inter-process communication, CORBA offers a powerful solution. TCP/IP endpoint negotiations in Blob Streaming are performed using CORBA messages. These negotiations usually contain small amounts of richly typed data, and therefore are well suited for CORBA.

The poor performance of CORBA bulk data transfer is a result of existing implementations that fail to optimize common sources of overhead. This overhead stems primarily from inefficient presentation layer conversions, data copying, memory management, and inefficient receiver-side demultiplexing and dispatching operations. This overhead is often masked on low-speed networks like Ethernet and Token Ring. On high-speed networks like ATM or FDDI, however, this overhead becomes a significant factor limiting communication performance [22]. To overcome these inefficiencies, we use sockets to setup point-to-point TCP connections and transmit bulk data efficiently across the connections. Since Blob Streaming does not interpret the data it transfers, the untyped nature of socket-level data exchange is acceptable.

Low-level network programming interfaces like sockets are hard to program because they have complex interfaces and are error prone. Our solution to this problem was to use C++ wrappers from the ACE toolkit [6] to encapsulate the C interfaces. ACE provides a rich set of efficient, reusable C++ wrappers, class categories, and frameworks that perform common communication software tasks (such as event demultiplexing, event handler dispatching, connection establishment, message routing, dynamic configuration of application services, and concurrency control).

It is important to note that ACE does not offer all the services of CORBA (such as object selection, location, activation, and parameter marshalling). Therefore, CORBA provides important value as a higher-level distributed object computing framework.

6 Concluding Remarks

We are currently deploying the Blob Streaming framework in a production distributed electronic medical imaging system being developed as part of Project Spectrum at the Electronic Radiology Lab (ERL) at the Washington University School of Medicine and BJC Health System, in collaboration with industrial partners Kodak Health Imaging Systems, IBM/ISSC, and Southwestern Bell Corporation. BJC is one of the nation's largest integrated health delivery systems, representing an alliance of health care partners in Missouri and southern Illinois.

Distributed electronic medical imaging systems like Project Spectrum require high-performance bulk data communication. The Blob Streaming framework described in this paper uses sockets to achieve high performance and uses CORBA to provide the flexibility needed for distributed electronic medical imaging systems. Blob Streaming allows application code to be developed independent of Blob location, Blob type, and Blob storage. These abstractions allow image processing algorithms to be reused for many types and locations of Blobs. In addition, Blob Streaming is designed to allow flexibility across platforms by abstracting from OS-specific mechanisms, concurrency policies, and event loops.

References

[1] G. Blaine, M. Boyd, and S. Crider, "Project Spectrum: Scalable Bandwidth for the BJC Health System," *HIMSS, Health Care Communications*, pp. 71–81, 1994.

[2] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[3] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[4] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), ACM, August 1996.

[5] A. Gokhale and D. C. Schmidt, "Performance of the CORBA Dynamic Invocation Interface and Internet Inter-ORB Protocol over High-Speed ATM Networks," in *Submitted to GLOBECOM '96*, (London, England), IEEE, November 1996.

[6] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the 6th USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[7] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," in *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)* (S. P. Berczuk, ed.), Wiley and Sons, 1996.

[8] D. C. Schmidt, "A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems," in *International Conference on Configurable Distributed Systems*, May 6–8 1996.

[9] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[11] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1994.

[12] D. D. Clark and D. L. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 200–208, ACM, Sept. 1990.

[13] J. Eykholt, S. Kleiman, S. Barton, R. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing... Multithreading the SunOS Kernel," in *Proceedings of the Summer USENIX Conference*, (San Antonio, Texas), June 1992.

[14] H. Custer, *Inside Windows NT*. Redmond, Washington: Microsoft Press, 1993.

[15] D. C. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, (Aarhus, Denmark), ACM, August 1995.

[16] A. Stepanov and M. Lee, "The Standard Template Library," Tech. Rep. HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[17] D. Ritchie, "A Stream Input–Output System," *AT&T Bell Labs Technical Journal*, vol. 63, pp. 311–324, Oct. 1984.

[18] D. C. Schmidt and T. Suda, "Transport System Architecture Services for High-Performance Communications Systems," *IEEE Journal on Selected Areas in Communication*, vol. 11, pp. 489–506, May 1993.

[19] K. Modeklev, E. Klovning, and O. Kure, "TCP/IP Behavior in a High-Speed Local ATM Network Environment," in *Proceedings of the 19th Conference on Local Computer Networks*, (Minneapolis, MN), pp. 176–185, IEEE, Oct. 1994.

[20] P. Software, *Quantify User's Guide*, 1995.

[21] G. Parulkar, D. C. Schmidt, and J. S. Turner, "a^It^Pm: a Strategy for Integrating IP with ATM," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, ACM, September 1995.

[22] M. DoVan, L. Humphrey, G. Cox, and C. Ravin, "Initial Experience with Asynchronous Transfer Mode for Use in a Medical Imaging Network," *Journal of Digital Imaging*, vol. 8, pp. 43–48, February 1995.

Class Relationships and User Extensibility in Solid Geometric Modeling

James R. Miller
University of Kansas
miller@eecs.ukans.edu

Abstract

Solid Geometric Modeling is an important enabling technology in Computer-Aided Design and Manufacturing. Open, extensible architectures which foster efficient construction and manipulation of models are important to design engineers. We describe the architecture of the cryph Solid Modeler, focusing on aspects of the design which maximize flexibility and enable user-extensibility of primitive modeling shapes.

1.0 Introduction

Geometric Modeling and Computer Graphics are important tools in modern industrial design and manufacture. Computer models are replacing physical models. They are cheaper to construct, easier to change, and simpler to analyze. They enable a broad range of automated technologies including finite element analysis, process planning, robotics, and computer-controlled manufacturing. Computer simulations save industry both time and money, and computer analyses of geometric models lead to better and cheaper products.

Geometric Modeling is concerned with efficient and robust computer-based representation and analysis of geometric information describing a product to be manufactured. An important sub-discipline of Geometric Modeling is Solid Modeling in which one creates a complete volumetric representation of an object [Requicha 80]. Geometric Modelers provide the capability to create and manipulate curves and surfaces. Solid Modelers build on top of this curve and surface functionality by establishing a framework in which the curve and surface elements are guaranteed to describe a valid, physically-realizable solid.

We describe our system from a programming language point of view, including its goals and architecture and some of our ongoing research efforts related to user extensibility of modeling primitives. No specialized knowledge of curve and surface representations is required to read this paper, however we do need to introduce a few basic concepts related to Solid Modeling. While several different solid representations have been studied over the years [Requicha 80], two schemes predominate in systems today: Constructive Solid Geometry (CSG) and Boundary Representations (B-Rep).

CSG representations define a solid as a general Boolean combination of half-spaces. Loosely speaking, a *half-space boundary* is a surface which divides space into two distinct regions: an "inside" and an "outside". Each region is called a *half-space*. For example, a plane is a half-space boundary which determines a half-space containing the points on one side of the plane and another half-space containing those points on the other side. Similarly, the half-spaces associated with a sphere are those points inside the sphere and those that are outside.

In CSG Solid Modeling, the designer rarely, if ever, thinks of the boundary of a desired half-space as distinct from the half-space itself. For example, a designer would not care to distinguish between creating a spherical surface and a spherical half-space. This distinction is one that is significant only to certain internal algorithms, and even at that, it is merely a matter of how the basic data defining the sphere is used. That is, the geometric definitions for the boundary and for the half-space are identical.

We mention these issues here since they affected the design of our class hierarchy. At issue were the sometimes conflicting demands of "specification purity" and "naturalness" of the user interfaces. We shall examine these issues in section 5.

B-Reps characterize a solid by an explicit description of its boundary (i.e., its "skin"). That is, they consist of a hierarchical collection of trimmed boundary elements (vertices, edges, and faces) which, when taken together, completely describe the exterior boundary of a solid. B-Rep data structures are typically designed to draw a sharp distinction between topological information (i.e., adjacency relationships which are independent of shape) and geometric information (i.e., the actual points, curves, and surfaces which determine shape).

Both CSG and B-Reps are prevalent because they can be effectively used to create a wide range of models, and because algorithms are known for most commonly desired analysis applications. In addition, these two representations have largely complementary advantages and disadvantages. Some systems -- called *dual representation* systems [Requicha 80, Miller 89] -- maintain both in a redundant fashion.

Today, Solid Modeling systems are growing in use, due largely to slow but steady progress in terms of numerical stability, geometric coverage, and application algorithms. Nevertheless, a great many technological challenges remain. Here we focus on two such issues related to programming language support:

- *controlled reuse and refinement of geometric software*

  Many operations applied to curves and surfaces depend only on the generic type of the entity. For example, the method for applying rigid transformations to conics is independent of the type of conic involved. On the other hand, simple robust intersection algorithms for intersecting two circles are preferred over more general curve-curve (or even conic-conic) intersection algorithms. Observations such as these have led many to the conclusion that object-oriented techniques including the application of inheritance with overriding can provide appropriate, natural, and powerful tools for the construction of Geometric Modeling systems.

- *user extensibility of modeling primitives*

  The designer of a modeling system must supply basic modeling and analytical tools supporting basic curve and surface definition as well as intersections and other algorithms. For creating solid models, a set of component solids is normally provided which a designer can use to construct a model. Loosely speaking, one either "cuts" with the component solid or "glues it onto" the evolving model. At the lowest level, these component solids are half-spaces, but specific common shapes (blocks, bounded portions of cylinders and cones, etc.) are often provided in addition to facilitate the engineer's work. This set of common shapes is unavoidably incomplete. There are inevitably application-specific "common shapes" which are not sufficiently general-purpose to warrant being a part of the standard product, yet they are of considerable value to specific groups of engineers. These engineers need to be able to tailor the standard system by adding such specialized component solids. Having done so, these specialized components must appear and behave exactly as those in the standard product.

The remainder of the paper is organized as follows. In the next section we discuss the relevant evolution of the *cryph* modeler which is being used as a testbed for techniques applicable to these problems, and we relate it to other possible approaches. The remaining sections discuss relevant portions of the cryph architecture from a programming language support point of view. In particular we discuss how and why various concepts were modeled as they were in C++. Finally we summarize in section 7.

2.0 The *cryph* Modeler and Other Engines

Cryph is a dual representation Solid Modeler developed at the University of Kansas based in part on curve and surface intersection utilities developed over the past five to ten years in ANSI C (e.g., [Miller 87, Miller & Goldman 95]). The original cryph modeler was written purely in ANSI C and used an implementation of a subset of PHIGS [ISO 89, Howard, et. al. 91] for display.

The version of cryph described here employs the same low-level curve and surface intersection utilities and boundary evaluator. However, development moved to C++ to enable a more flexible representation for the geometry[1]. In particular, the CSG representation was explicitly designed to enable the user extensibility features discussed in section 4 which were the focus of much of the original effort. In fact, the original structure of the boundary evaluator [Miller 93] was designed specifically to provide low-level support for such extensibility. The current version also uses OpenGL [Neider, et. al. 93] and the object-oriented OpenInventor [Wernecke 94] for graphics display and interaction.

[1] Interesting issues arose while determining how to marry the old utilities with the new class definitions, but a complete discussion of this history is beyond the scope of the current presentation.

From a solid modeling point of view, the primary type of user-definable extension we wanted to provide was the ability to create application-specific "macro solids" which would appear exactly like system-supplied instantiable solids. These are fairly straightforward to create with the architecture we now have, yet they are surprisingly powerful and useful.

Another type of extension would be to allow users to add new surface types (and hence also new curve types). While possible in principle, this remains an extraordinarily complex task. Adding macro solids requires only a packaging of a suitably parameterized CSG sub-tree; adding new surface types, on the other hand, minimally requires that new surface-surface and curve-surface intersection algorithms be provided by the person wishing to add these new geometric forms. Not only does this involve considerably more sophisticated software development, but also we have not yet addressed the issue of documenting to such a developer all the issues involved in adding such new forms. Indeed, there are research issues lurking there which need to be addressed [Lamping 93][2].

The current (C++) version of the cryph modeler views the older cryph utilities as a solid modeling engine. This concept is not unique. In recent years, much effort has been expended in the specification and prototyping of such potential engine interfaces. Computer-Aided Manufacturing, International (CAM-I) was an early leader in this effort [CAM-I 90]. More recently, Spatial Technology's ACIS modeler and toolkit [ACIS 95] has emerged as a popular engine for a variety of modeling systems.

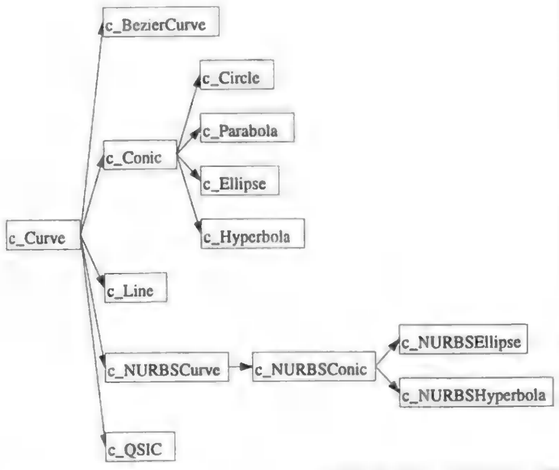
While ACIS supports Boolean operations between solids, it is basically a pure B-Rep system. A major thrust of our overall research agenda revolves around the potential of dual CSG/B-Rep modeling, and we needed an engine with support for such dual models at the core. This was a major reason we continued to use the older cryph utilities as opposed to the commercial ACIS product.

Other differences of interest between the two engines relate to the use of Nonuniform Rational B-Splines (NURBS). In cryph, NURBS curves and surfaces are used only for the interface to OpenGL and OpenInventor. That is, NURBS objects are created on the fly from the cryph B-Rep when needed for display. In ACIS, NURBS curves and surfaces are a basic modeling form. A detailed discussion of the related implications is beyond the scope of this paper.

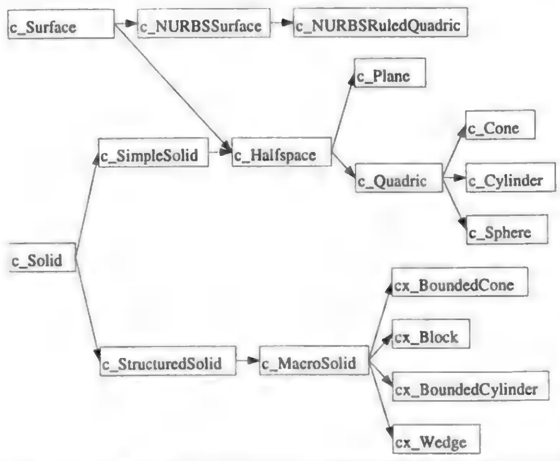
[2] In our case, this is also complicated in no small part by the ANSI C heritage of the curve and surface intersection utilities.

3.0 Substratum Curve and Surface Issues

Lines, planes, conics, and quadrics are currently the core geometric forms in cryph. The relationships between these classes are depicted in Figure 1.



(a) Curves



(b) Surfaces and Solids

Figure 1: The cryph Curve, Surface, and Solid Class Hierarchies

These portions of the class hierarchy reflect the usual goals of factoring code for common operations while providing for specialization of operations in particular cases.

As implied earlier, however, the situation is different for Bezier and NURBS curves and surfaces since they are only generated as alternate representations of parabolas (Bezier) and other conics and quadrics (NURBS) to facilitate the graphical display interface. The subclasses of these freeform shapes are used to record how instances were created so that subsequent modification and query operations can be facilitated.

Sample class definitions for a vertical slice of the *c_Curve* hierarchy (in particular, the *c_Curve⇒c_Conic⇒c_Ellipse* classes) are presented in Figure 2 (a-c). The data type "*real*" is platform-dependent, but is typically defined to be double precision. The classes "*aPoint*" and "*aVector*" encapsulate the notions of points and vectors in three-dimensional affine space.

```
class c_Curve
{
public:
    c_Curve();
    c_Curve(const c_Curve& c);

    virtual ~c_Curve();

    // instance methods
    virtual void  ASCIIdisplay(ostream& os)
            const = 0;
    virtual real  getParameterOf(const aPoint& p)
            const = 0;
    virtual void  getPointAt(real parameter,
            aPoint& p) const = 0;
    virtual int   pwlApproximation(real errorHint,
        aPoint       outputBuffer[],
        int          outputBufferLength,
        orientation  o=UNSPECIFIEDorientation,
        endPointSpec* endPt = NULL)
                { return 0; }
    virtual const char* const
                typeName() const = 0;

    // operators
    c_Curve&   operator=(const c_Curve& rhs);
};
```

Figure 2(a): The *c_Curve* class

To support fundamental solid modeling operations, it is necessary to compute intersections (i.e., find the set of points common to) various combinations of instances of curves and surfaces. Most generally we require pairwise intersections; occasionally it is desirable to find the set of points common to three surfaces simultaneously. The result of a general intersection operation (maintained in an instance of class *c_IntersectionResult*) is either empty (i.e., the operands share no points), or it consists of some collection of discrete points and (portions of) curves and surfaces.

```
class c_Conic : public c_Curve
{
public:
    c_Conic();
    c_Conic(const c_Conic& C);

    c_Conic(const aPoint& O , const aVector& U,
                const aVector& W);
    c_Conic(const aPoint& O ,
                const aVector& W);

    virtual ~c_Conic();

    void      getLocalCoordinates(const aPoint& p,
                real& x, real& y) const;
    aPoint    getOrigin() const;
    aVector getU() const;
    aVector getV() const;
    aVector getW() const;

    // Class Methods

    static    void putThetaInStandardRange(
                real& theta);

    // OPERATORS
    virtual c_Conic& operator=(const c_Conic&);

protected:
    // All conics have a local coordinate system:
    aPoint      mOrigin;
    aVector     mU;
    aVector     mV;
    aVector     mW;
private:
    void        validateData();
};
```

Figure 2(b): The *c_Conic* sub-class

It is our practice to invoke specialized algorithms based on the type of curves and surfaces given to these utilities. The lack of reasonable support in C++ to support such double (and occasionally triple) dispatch was an early disappointment. We were at least able to hide the awkwardness of the actual dispatch from the programmer by modeling requests for intersections as constructors for class *c_IntersectionResult*:

c_IntersectionResult(const c_Curve& c1, const c_Curve& c2);
c_IntersectionResult(const c_Curve& c1, const c_Surface& s2);
c_IntersectionResult(const c_Surface& s1, const c_Surface& s2);
c_IntersectionResult(const c_Surface& s1, const c_Surface& s2, const c_Surface& s3);

The basic idea is that when the system needs to compute the intersection of some combination of curves and surfaces, the programmer declares an instance of *c_IntersectionResult*, specifying the required curves and surfaces as implied above. Depending on the dynamic classes of the curves and surfaces, specialized algorithms are invoked to compute the various pieces of the intersection and store them in instances of class *c_IntersectionItem*. The resulting collection of these items is then associated with the *c_IntersectionResult*. This approach allowed us to minimize the size and complexity of the class interfaces for both the curves and surfaces as well as for the intersection result itself.

4.0 Basic Solid Modeling and User Extensibility

We define two fundamental subclasses of solids: "simple" and "structured". (See Figure 1(b).) Simple solids are half-spaces, the base objects to which the CSG Boolean operations are applied. Structured solids are general Boolean combinations of half-spaces. They are the only types of solids for which it is useful to keep an associated B-Rep. In our system, B-Reps can be associated with each instance of a structured solid in a CSG graph. However in practice B-Reps are normally maintained only at the top one or two levels due to storage considerations.

Simple and structured solids differ also in how they are generated by the designer and validated by the system. Simple solids (half-spaces) can be instantiated via numerical parameters which are simple and orthogonal. That is, each parameter can be tested for validity independent of all others. If an invalid parameter is detected, it is easy to substitute a reasonable default value which will yield a valid instance. Structured solids, on the other hand, are generated by specifying Boolean operations between simple or other structured solids. It is impossible to generate an "invalid" structured solid using the standard operators.

```
class c_Ellipse : public c_Conic
{
public:
    c_Ellipse();
    c_Ellipse(const c_Ellipse& E);
    c_Ellipse(const c_Circle& C);

    c_Ellipse(const aPoint& C ,
          const aVector& U, const aVector& W,
          real Rmajor, real Rminor);

    virtual ~c_Ellipse();

    // OPERATORS
    friend   ostream& operator<<(ostream& os,
                   const c_Ellipse& obj);
    virtual c_Ellipse& operator=(
                   const c_Ellipse&);

    // GENERAL METHODS
    virtual  void     ASCIIdisplay(ostream& os)
                           const;
             aPoint   getCenter() const;
             real     getMajorRadius() const;
             real     getMinorRadius() const;
             real     getParameterOf(
                           const aPoint& p) const;
             void     getPointAt(real parameter,
                           aPoint& p) const;
    virtual  int pwlApproximation(real errorHint,
         aPoint       outputBuffer[],
         int          outputBufferLength,
         orientation  o=UNSPECIFIEDorientation,
         endPointSpec* endPt = NULL);
    virtual const char* const
                      typeName() const;

    // CLASS METHODS
    static const char* const  className();
private:
    static const char* const
             TypeName;
    void     validateData();

    real     mMajorRadius;
    real     mMinorRadius;
};
```

Figure 2(c): The *c_Ellipse* sub-class

4.1 User Extensibility via "Macro Solids"

A subclass of structured solids -- *c_MacroSolid* in Figure 1(b) – re-introduces instantiation via numerical parameters. It is here where users extend the set of modeling primitives as mentioned in the Introduction.

To recap, the motivation was based on simplifying the work required of the design engineer. That is, the set of half-spaces coupled with the Boolean operators is sufficient to generate desired models. However many geometric features appear with great regularity, and it becomes horribly inefficient to recreate each instance with appropriate instantiations and combinations of half-spaces.

Of course this has been known for almost as long as solid modelers have been in existence; hence since the 1970s most developers of solid modelers have provided an additional set of such shapes which can be directly instantiated. Typical simple examples include a bounded cylinder (the portion of a cylindrical half-space between two given planar half-space boundaries), and a parallelepiped (the intersection of six planar half-spaces).

The fact that these common shapes are really just "macros" for certain common construction sequences is not apparent in most systems, however. In many instances this is due to how the boundary evaluation algorithms work. Many need to be bootstrapped from simple bounded objects whose boundary topology is known a priori. In addition, system designers typically implemented these shapes as specific built-in objects in closed systems, and that meant that it was not viable for end-users to augment the system-supplied set with some of their own favorites. When such extensions were allowed by the system, the resulting CSG tree would typically have a large number of duplicate half-spaces.

The boundary evaluator in cryph was specifically designed to work directly from half-spaces for exactly this reason [Miller 95], and we have provided a standard interface in the class hierarchy where users can add their own set of specialized shapes, created with minimally-sized CSG trees. Such shapes are added by defining an appropriate subclass of *c_MacroSolid* in which a particular set of methods is defined and implemented. We used this facility ourselves to implement the pseudo-standard set of such shapes: bounded cone, block (parallelepiped), bounded cylinder, and wedge (half of a parallelepiped). These are illustrated in Figure 1(b). In addition, students in our Geometric Modeling class have defined and used their own as part of class projects using this system.

A necessary evil associated with macro solids is that the actual instantiation and validation of instances of these specialized modeling shapes can be more complex than that for half-spaces. This is due largely to the fact that, unlike half-spaces, the instantiation parameters for macro solids are typically *not* orthogonal, and hence the validity of a given parameter cannot in general be decided in isolation from others. For example, the parallelepiped primitive takes three vectors specifying the directions of the edges, and the validity condition is that these three vectors be linearly independent. When instantiation errors are detected, it is therefore often more difficult to determine how the situation can best be corrected.

5.0 On Multiple Inheritance

As illustrated in Figure 1(b), class *c_Halfspace* inherits multiply from *c_Surface* and *c_SimpleSolid*, modeling the idea that half-spaces can be used for surface operations such as intersections as well as for solid operations such as deciding whether a point lies in the interior of a solid. Modeling the classes in this way contradicts what we described in the Introduction, namely that half-space boundaries are surfaces which separate the points in two distinct half-spaces. Thus in the interests of specification purity, one should define a class "half-space-boundary" which would inherit from *c_Surface*, and then model "half-space" as a subclass of *c_SimpleSolid*. The relationship between these two would then be "half-space *has-a* half-space-boundary".

While more accurate from a mathematical point of view, there seemed to be a danger that the user interface could be complicated. As a quick example, if the designer wants to create a sphere, we would need to determine (i.e., the designer would have to tell us) whether a spherical skin or a spherical solid were actually desired. This would require first that we explain to the design engineer that there is a difference between half-spaces and half-space boundaries, a tenuous proposition.

Moreover, like us, the designer often wants to create such an object once and use it both ways. For example, he may create two spheres, ask whether some point is inside both, and then ask for their circle of intersection. The designer typically asks such questions without thinking about whether it is a question asked of a solid or of a surface.

Since the defining data is identical for both, forcing such a seemingly arbitrary distinction would most likely be met with disdain.

We chose to define *c_Halfspace* using multiple inheritance since the difference between half-spaces and half-space boundaries is of no interest to engineers, and since there is never any internal ambiguity. That is, it is always clear whether surface properties or solid properties are of interest when a given method is invoked. Furthermore, there is no overhead in terms of instance variables since the defining data for both is identical.

6.0 Rendering

In contrast to the distinctions considered in the previous section, it is vitally important to distinguish between a *geometric model* and an *image* of the model. While the concept of a model is independent of any display device or graphics system, images are intimately tied to these. In addition, images often involve various geometric approximations and/or re-representations, as well as other intentional visual distortions. We often also wish to support multiple simultaneous visual representations of an object.

All these goals dictate that view-independent geometric descriptions be maintained which know nothing about graphical display. One or more separate class hierarchies can then be developed which support particular display approaches. Instances of these "renderers" are created for individual entities to be displayed. The renderer knows what is required of a particular visualization medium, and it knows how to ask appropriate questions of the geometry in order to obtain data to drive the graphics engine.

An example of a "renderer" class hierarchy used in cryph is illustrated in Figure 3. Instances of these renderers interface to the object-oriented OpenInventor engine on Silicon Graphics workstations (and occasionally directly to OpenGL in addition) to produce interactive visualizations of the cryph B-Rep (Figure 4).

Notice that the renderer hierarchy does not match the application class hierarchy. This should not be surprising. Consider the following:

- Different characteristics are important in the geometric model hierarchy than are important in the renderer hierarchy. An obvious example is the fact that the renderers care first about whether the object to be rendered is bounded or unbounded.
- The geometric model hierarchy is also much deeper. This is in part to factor common analysis code and to facilitate simple, yet flexible interface specifications (for example, the *c_Curve⇒c_Conic⇒c_Ellipse* relationships), and in part to record *how* certain instances of an object were created in order to facilitate construction, modification, and other query operations (e.g., the *c_NURBSCurve ⇒ c_NURBSConic ⇒ c_NURBSEllipse* relationships).

  Generally speaking, there is an *i_Renderer* subclass for every leaf class in the *c_Curve*, *c_Surface*, and *c_Solid* hierarchies. The exceptions are where subclasses are employed only to record *how* certain instances of an object were created (i.e., NURBS curves and surfaces and *c_StructuredSolid*). The renderer is only concerned with the object, not how it was created.
- Multiple inheritance was not needed on the renderer side, but recall it was used on the application side to model certain application-specific *and image-independent* relationships involving half-spaces. In our work to date, we have only required visualizations of the boundaries of objects, hence the fact that a half-space can also be interpreted as a solid has been irrelevant to the renderer. In fact, the current solid renderer in effect simply extracts the outer skin of the solid piece by piece, passing each piece to individual surface renderers.

  Techniques for volumetric visualization through direct volume rendering are well-known [Watt & Watt 92]. They are extremely useful when attempting to visualize certain types of internal structure and/or the distribution of functional properties (heat, stress, etc.) which vary continuously throughout the interior of an object.

  As our work progresses, it is very likely that we will require such volumetric visualization capabilities. However, it is unlikely that multiple inheritance will be employed at that time since the rendering techniques for surfaces and volumes are radically different. Thus no argument can be made that volumetric renderers have sufficient common functionality with surface renderers to warrant an inheritance relationship.

Unlike the arguments we made when discussing the definition of classes for half-spaces, no awkwardness would be foisted upon either the designer or the programmer since the class to be created depends only on the type of visualization desired (something which obviously must be specified anyway) and such a renderer would never be expected to play a "dual role". That is, we would never expect such a renderer to generate a more traditional surface display.

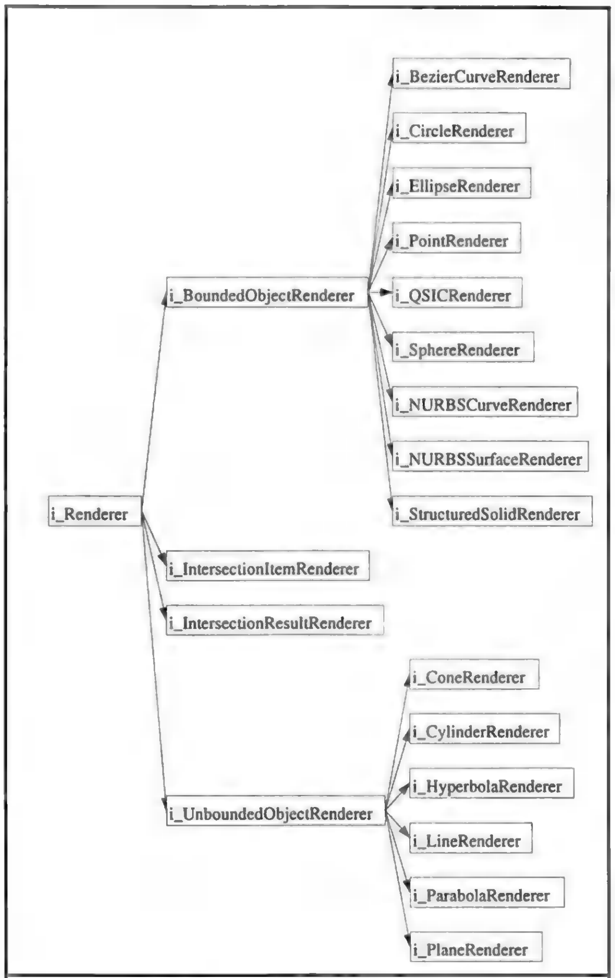


Figure 3
A Renderer Class Hierarchy

We have additional visualization interfaces which output text files for use by other visualization programs. For example, we generate POV-Ray input files [POV 93] to generate ray traced images like those in Figure 5. These interfaces are currently implemented as methods in the geometric classes. They add negligible overhead and do not compromise the integrity of the model in any way. That is, they simply dump textual data describing the model to an *fstream*.

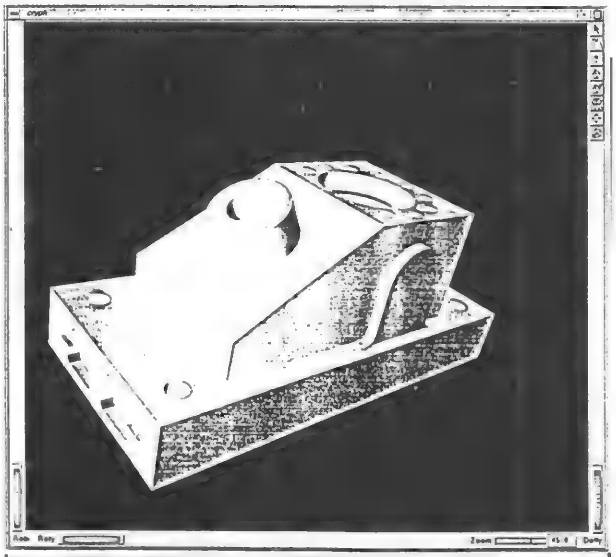


Figure 4
An OpenInventor Display of a B-Rep Solid

7.0 Summary

We have described the history and basic architecture of the cryph solid geometric modeling system, focusing on how the class hierarchies were designed in order to satisfy various important design criteria related to user extensibility and graphics system portability. We have also presented our rationale for the use of multiple inheritance in the surface and solid portions of the hierarchy, observing that our use of multiple inheritance was driven by user interface considerations and internal expediency, but was not immune to criticism. Finally we described the use of separate class hierarchies to support visualization of and interaction with the model in a way that keeps the notion of *model* distinct from *image of a model.*

To date we have focused on the construction of the infrastructure of the solid modeler. We are currently pursuing projects which use this engine in a number of ways. We are working on an interactive modeler which, among other things, will allow the designer to create new macro solids interactively. For example, the modeler would be used to construct the shape and to specify relevant parametric relationships. The system would then automatically generate C++ definitions and code for an appropriate *c_MacroSolid* subclass. We are also investigating ways to construct more sophisticated metaclasses which would allow new macro solids to be created on the fly without the need for generation of actual new C++ subclasses.

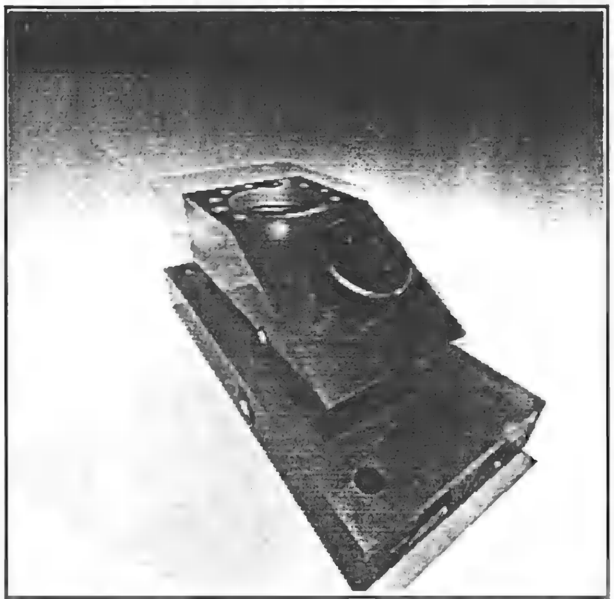


Figure 5: A Ray Traced Image of a CSG Solid

A generalization of these efforts is being studied which focuses on the use of the engine to support variational classes of models [Shapiro & Vossler 95]. A number of additional issues arise in that context, not the least significant of which is the need for a mechanism to record and subsequently solve sets of quite general constraint equations which describe the geometric relationships of importance in a variational family of parts.

Acknowledgments

Much of the work described here was performed in DesignLab, a multidisciplinary research laboratory funded in part by NSF Grant CDA-94-01021.

References

[ACIS 95]
Spatial Technology, *ACIS*, http://www.spatial.com/spatial.

[CAM-I 90]
Computer-Aided Manufacturing, International, *Application Interface Specification (AIS) 2.0*, CAM-I, Arlington, Texas.

[Howard, et. al. 91]
T. Howard, W. Hewitt, R. Hubbold, and K. Wyrwas, *A Practical Introduction to PHIGS and PHIGS PLUS*, Addison-Wesley, 1991.

[ISO 89]
International Organization for Standardization, *ISO 9592: Programmer's Hierarchical Interactive Graphics System (PHIGS)*, 1989.

[Lamping 93]
J. Lamping, Typing the Specialization Interface, *ACM SIGPlan Notices (Proceedings OOPSLA '93)*, Vol. 28, No. 10, October 1993, pp. 201-214.

[Miller 87]
J. R. Miller, Geometric Approaches to Nonplanar Quadric Surface Intersection Curves, *ACM Transactions on Graphics*, Vol. 6, No. 4, October 1987, pp. 274-307.

[Miller 89]
J. R. Miller, Architectural Issues in Solid Modeling, *IEEE Computer Graphics and Applications*, Vol. 9, No. 5, September 1989, pp. 72-87.

[Miller 93]
J. R. Miller, Incremental Boundary Evaluation Using Inference of Edge Classifications, *IEEE Computer Graphics and Applications*, Vol. 13, No. 1, January 1993, pp. 71-78.

[Miller 95]
J. R. Miller, Incremental Boundary Evaluation For Nonmanifold Partially Bounded Solids, *Graphics Interface '95*, Québec City, 15-19 May 1995, pp. 187-195.

[Miller & Goldman 95]
J. R. Miller and R. N. Goldman, Geometric Algorithms for Detecting and Calculating All Conic Sections in the Intersection of Any Two Natural Quadric Surfaces, *Computer Vision, Graphics, and Image Processing*, Vol. 57, No. 1, January 1995, pp. 55-66.

[Neider, et. al. 93]
J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide*, Addison-Wesley, 1993.

[POV 93]
POV-Ray Team, *Persistence of Vision Ray Tracer (POV-Ray)*, Version 2.0, User's Documentation, 1993.

[Requicha 80]
Representations for Rigid Solids: Theory, Methods, and Systems, *ACM Computing Surveys*, Vol. 12, No. 4, December 1980, pp. 437-464.

[Shapiro & Vossler 95]

V. Shapiro and D. Vossler, What is a Parametric Family of Solids?, *Proceedings, Third Symposium on Solid Modeling and Applications*, Salt Lake City, ACM Press, May 1995, pp. 43-54.

[Watt & Watt 92]
A. Watt and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley, 1992.

[Wernecke 94]
J. Wernecke, *The Inventor Mentor*, Addison-Wesley, 1994.

A Distributed Object Model for the Java™ System

Ann Wollrath, Roger Riggs, and Jim Waldo
JavaSoft
{ann.wollrath, roger.riggs, jim.waldo}@sun.com

Abstract

We show a distributed object model for the Java™[1] System [1,6] (hereafter referred to simply as "Java") that retains as much of the semantics of the Java object model as possible, and only includes differences where they make sense for distributed objects. The distributed object system is *simple*, in that a) distributed objects are easy to use and to implement, and b) the system itself is easily extensible and maintainable. We have designed such a model and implemented a system that supports remote method invocation (RMI) for distributed objects in Java. This system combines aspects of both the Modula-3 Network Objects system [3] and Spring's subcontract [8] and includes some novel features.

To achieve its goal of seamless integration in the language, the system exploits the use of *pickling* [14] to transmit arguments and return values and also exploits unique features of Java in order to dynamically load stub code to clients[2]. The final system will include distributed reference-counting garbage collection for distributed objects as well as lazy activation [11,16].

1 Introduction

Distributed systems require entities which reside in different address spaces, potentially on different machines, to communicate. The Java™ system (hereafter referred to simply as "Java") provides a basic communication mechanism, sockets [13]. While flexible and sufficient for general communication, the use of sockets requires the client and server using this medium to engage in some application-level protocol to encode and decode messages for exchange. Design of such protocols is cumbersome and can be error-prone.

1. Java and other Java-based names and logos are trademarks of Sun Microsystems, Inc., and refer to Sun's family of Java-branded products and services.
2. Patent pending

An alternative to sockets is Remote Procedure Call (RPC) [13]. RPC systems abstract the communication interface to the level of a procedure call. Thus, instead of application programmers having to deal directly with sockets, the programmer has the illusion of calling a local procedure when, in fact, the arguments of the call are packaged up and shipped off to the remote target of the call. Such RPC systems encode arguments and return values using some type of an external data representation (e.g., XDR).

RPC, however, does not translate well into distributed object systems where communication between program-level *objects* residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require *remote method invocation* or RMI. In such systems, the programmer has the illusion of invoking a method on an object, when in fact the invocation may act on a remote object (one not resident in the caller's address space).

In order to support distributed objects in Java, we have designed a remote method invocation system that is specifically tailored to operate in the Java environment. Other RMI systems exist (such as CORBA) that can be adapted to handle Java objects, but these systems fall short of seamless integration due to their inter-operability requirement with other languages. CORBA presumes a heterogeneous, multi-language environment and thus must have a language neutral object model. In contrast, the Java language's RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore follow the Java object model whenever possible.

We identify several important goals for supporting distributed objects in Java:

- support seamless remote invocation between Java objects in different virtual machines;
- integrate the distributed object model into the Java language in a natural way while retaining most of Java's object semantics;

- make differences between the distributed object model and the local Java object model apparent;
- minimize complexity seen by the clients that use remote objects and the servers that implement them;
- preserve the safety provided by the Java runtime environment.

These goals fall under two main categories: the simplicity and naturalness of the model. It is most important that remote method invocation in Java be simple (easy to use) and natural (fit well in the language).

In addition, the RMI system should perform garbage collection of remote objects and should allow extensions such as server replication and the activation of persistent objects to service an invocation. These extensions are transparent to the client and add minimal implementation requirements on the part of the servers that use them. These additional features motivate our system-level goals. Thus, the system must support:

- several invocation capabilities
  - simple invocation (unicast)
  - invocation to multicast groups (to enable server replication)
  - extensibility to other invocation paradigms
- various reference semantics for remote objects
  - live (or non-persistent) references to remote objects
  - persistent references to and lazy activation of remote objects
- the safe Java environment provided by security managers and class loaders
- distributed garbage collection of active objects
- capability of supporting multiple transports

In this paper we will briefly describe the Java object model, then introduce our distributed object model for Java. We will also describe the system architecture and relevant system interfaces. Finally, we discuss related work and conclusions.

2 Java Object Model

Java is a strongly-typed object-oriented language with a C-style syntax. The language incorporates many ideas from languages such as Smalltalk [5], Modula-3 [10], Objective C [12] and C++ [4]. Java attempts to be simple and safe while presenting a rich set of features in the object-oriented domain.

Interfaces and Classes

One of the interesting features of Java is its separation of the notion of interface and class. Many object-oriented languages have the abstraction of "class" but provide no direct support (at the language level) for interfaces.

An *interface*, in Java, describes a set of methods for an object, but provides no implementation. A *class*, on the other hand, can describe as well as implement methods. A class may also include *fields* to hold data, but interfaces cannot. Thus, a class is the implementation vehicle in Java; an interface provides a powerful abstraction that contains no implementation detail.

Java allows subtyping of interfaces and classes by the use of *extension*. An interface may extend one or more interfaces; this capability is known as multiple-inheritance. Classes, however, are single-inheritance and may extend at most one other class.

While a class may extend at most one other class, it may *implement* any number of interfaces. A class that implements an interface provides implementations for all the methods described in that interface. If a class is defined to implement an interface, but does not provide an implementation for a particular method of that interface, it must declare that method to be *abstract*. A class containing abstract methods may not be instantiated.

An example of an arbitrary class definition in Java is as follows:

```
class Bar
     extends Foo
     implements Ping, Pong { ... }
```

where Bar is the class name, Foo is the name of the class being extended and Ping and Pong are the names of interfaces implemented by the class Bar.

Object Class Methods

All classes in Java extend the class Object, either implicitly or explicitly. The class Object has several methods which an extended class can override to have behavior specific to that class. These methods are:

- equals — tests the argument for equality with the object
- hashCode — returns a hash code for the object
- toString — returns a string representing the object
- clone — returns a clone of the object
- finalize — called to allow cleanup when the object is garbage collected

These methods are integral to the semantics of objects in Java.

Method Invocation

Method invocation in Java has the following syntax:

```
result = object.method(arg1, arg2, ...);
```

where: object is the entity which is being acted upon, method is the name of the method being called, argN is a parameter to the method, and result is the return value.

Method Parameters and Return Values

In Java, all parameters to and return values from a method are passed *by-value*. Only *references* to objects exist in Java, so object references (not objects) are passed by value. Thus, a change to an object passed to a method will be visible to the caller of the method.

The type of an object passed polymorphically does not change the type of the underlying object.

3 Distributed Object Model

In our model, a *remote object* is one whose methods can be accessed from another address space, potentially on a different machine. An object of this type is described by a *remote interface*, which is an interface (in Java) that declares the methods of a remote object. Remote method invocation (or RMI) is the action of invoking a method (of a remote interface) on a remote object. Most importantly, a method invocation on a remote object has the same syntax as a method invocation on a local object.

Clients of remote objects program to remote interfaces, not to the implementation classes of those interfaces. Since the failure modes of accessing remote objects are inherently different than the failure semantics of local objects, clients must deal with an additional exception that can occur during any remote method invocation.

What follows is a brief comparison of the distributed object model and the Java object model. The similarities between the models are:

- a reference to a remote object can be passed as an argument or returned as a result in any method invocation (local or remote);
- a remote object can be cast to any of the set of remote interfaces supported by the implementation using the built-in Java syntax for casting;
- the built-in Java instanceof operator can be used to test the remote interfaces supported by a remote object.

There are several basic differences between the distributed object model and the Java object model:

- clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces;
- clients must handle an additional exception for each remote method invocation;
- parameter passing semantics are slightly different in calls to remote objects;
- semantics of Object methods are defined to make sense for remote objects.

Remote Interfaces

In order to implement a remote object, one must first define a remote interface for that object. A remote interface must extend (either directly or indirectly) a distinguished interface called java.rmi.Remote. This interface is completely abstract and has no methods.

```
interface Remote {}
```

For example, the following code fragment defines a remote interface for a bank account that contains methods that deposit to the account, withdraw from the account, and get the account balance:

```
import java.rmi.*;

public interface BankAccount
        extends Remote
{
     public void deposit(float amount)
          throws RemoteException;
     public void withdraw(float amount)
          throws OverdrawnException,
                 RemoteException;
     public float balance()
          throws RemoteException;
}
```

As shown above, each method declared in an interface for a remote object must include java.rmi.RemoteException in its throws clause. If RemoteException is thrown during a remote call, then some communication failure happened during the call. Remote objects have very different failure semantics than local objects. These failures cannot be hidden from the programmer since they cannot be masked by the underlying system [15]. Therefore, we choose to expose the additional exception RemoteException in all remote method calls, so that programmers can handle this failure appropriately.

Remote Implementations

There are two ways to implement a remote interface (such as BankAccount). The simplest implementation route is for the implementation class, e.g., BankAcctImpl, to *extend* the class RemoteServer. We call this first scheme *remote implementation reuse*. Figure 1 below is an illustration of the interface and class hierarchies for remote interfaces and implementations in this scheme.

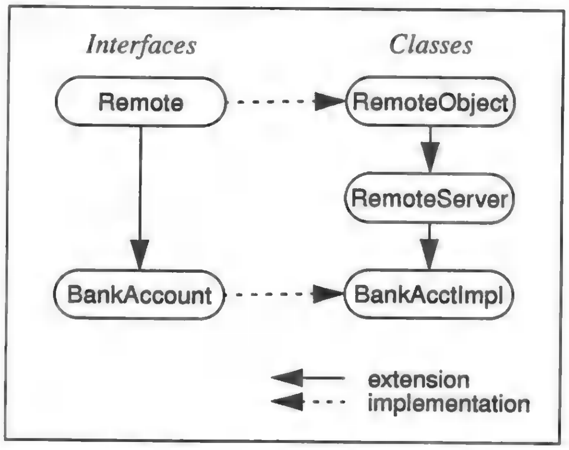


Figure 1. Reusing a remote implementation

The default constructor for RemoteServer takes care of making an implementation object remotely accessible to clients by *exporting* the remote object implementation to the RMI runtime. The class RemoteObject overrides methods inherited from Object to have semantics that make sense for remote objects. We will discuss what the appropriate semantics for these methods are in the section on "Object Method Semantics".

In the second implementation scheme, called *local implementation reuse*, the implementation class for a remote object does not extend RemoteServer but may extend any other local implementation class as appropriate. However, the implementation must explicitly export the object to make it remotely accessible.

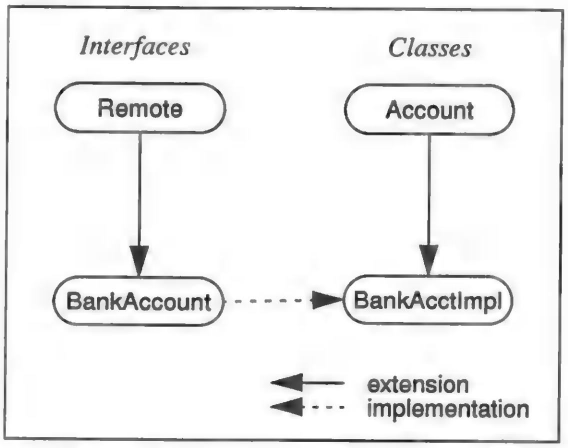


Figure 2. Reusing a local implementation class

The "local implementation reuse" scheme (shown in Figure 2), while allowing the class to reuse existing implementation code, does require that the class deal with the details of making instances of that class remotely accessible (by exporting the object to the RMI runtime). Such exporting is already taken care of in the RemoteServer constructor used in the first scheme.

Implementations using the second scheme must also be responsible for their own Java Object semantics and therefore must redefine methods inherited from the class Object appropriately. These object methods are already taken care of in the implementation of RemoteObject, used in the other scheme.

We deem the "remote implementation reuse" scheme more seamlessly integrated into the Java object model as well as requiring less implementation detail; so we will explain that implementation scheme in depth here. The other scheme, local implementation reuse, is included for implementation flexibility if such is required by the programmer.

Thus, BankAcctImpl, an implementation class of the remote interface BankAccount can be defined by extending RemoteServer as follows and would implement all the methods of BankAccount:

```
package myPackage;

import java.rmi.RemoteException;
import java.rmi.server.RemoteServer;

public class BankAcctImpl
        extends RemoteServer
        implements BankAccount
{
    public void deposit(float amount)
         throws RemoteException {...};
    public void withdraw(float amount)
         throws OverdrawnException,
                RemoteException {...};
    public float balance()
         throws RemoteException {...};
}
```

A few additional notes about implementing remote interfaces are:

- An implementation class may implement any number of remote interfaces.
- An implementation class may extend any other implementation class of a remote interface.
- *Only* those methods that appear in a remote interface (one that extends Remote either directly or indirectly) can be accessed remotely; thus non-remote methods in an implementation class can only be accessed locally.

The server implementation scheme fits very well into the Java object model and the Java language.

Remote Reference Types

In the distributed object model, clients interact with stub (surrogate) objects that have *exactly* the same set of remote interfaces defined by the remote object's class; the stub class does not include the non-remote portions of the class hierarchy that constitutes the object's type graph. This is because the stub class is generated from the most refined implementation class that implements one or more remote interfaces. For example, if C extends B and B extends A, but only B implements a remote interface, then a stub is generated from B, not C.

Because the stub implements the same set of remote interfaces as the remote object's class, the stub has, from the point of view of the Java system, the same type as the remote portions of the server object's type graph. A client, therefore, can make use of the built-in Java operations to check a remote object's type and to cast from one remote interface to another, e.g.:

```
Remote obj = ...;// lookup object
if (obj instanceof BankAccount) {
     BankAccount acct = (BankAccount)obj;
     //...
}
```

The system employs a mechanism called *dynamic stub loading* to make the correct stub for the remote object available to the client (this technique is fully described in the section "System Architecture").

Remote Method Invocation

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is obtained in the usual manner: as a return value in a method call or as a parameter passed to a method. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts.

Invoking a method on a remote object has the same syntax as invoking a method on any Java object. For example, here's how the bank account could be accessed (without exception handling):

```
BankAccount acct = ...;// lookup account
float balance;
acct.deposit(243.50);
acct.withdraw(100.00);
balance = acct.balance();
```

Since remote methods include RemoteException in their signature, the caller must be prepared to handle those exceptions in addition to other application specific exceptions. So, for each of the calls above (deposit, withdraw, and balance), the code needs to catch RemoteException (and the withdraw call would need to also catch OverdrawnException).

If RemoteException is thrown during a remote call, then some communication failure happened during the call. The client has little to no information on the outcome of the call—whether a failure happened before, during, or after the call completed. Thus, remote interfaces should be designed with these failure semantics in mind [9,15]. The semantics of a remote method may need to be idempotent whereas calls within the local address space likely do not have to be. Note that the above bank account interface does not support idempotent operations, so if an operation fails, the client needs to perform some type of recovery to determine the true state of the bank account (using transactions would solve this problem).

In most cases, a method invoked on a remote object is indirected through the remote object's stub to which the caller has a reference. In a method invocation to a remote object which actually *resides* in the same virtual machine as the caller, the call *may* be a local invocation and not a call via the stub for the remote object. If the caller has an actual reference to the remote object implementation, the method call is local and is not forwarded via a stub. However, a caller may receive, from a remote object in a different virtual machine, a remote reference to the object whose implementation is in the same virtual machine. In this case, the client (the caller) has a reference to a stub for the remote object; thus, a method call on this reference would be indirected through the stub.

Object Method Semantics

The default implementations for the methods of class Object (equals, hashCode, toString, clone, and finalize) are not appropriate for remote objects. The class RemoteObject provides implementations for these methods that have semantics more appropriate for remote objects.

In order for a remote object to be used as a key in a hash table, the methods equals and hashCode need to be overridden in a remote object implementation. The semantics of equals for a remote object must be defined such that remote objects have *reference equality*. Thus, given any two remote references to the same underlying object, those objects will be equal. No stronger equality, such as "content" equality, may be defined for remote objects, since determining the equality of contents would require a remote call. Remember that in a remote call, a RemoteException may be raised, and the method equals has no such exception in its throws clause. Due to the different failure semantics between local and remote calls, we chose to implement only reference equality for remote objects.

The hashCode method will return the same value for remote references that refer to the same underlying object.

The toString method is defined to return a string which represents the reference of the object. In the current implementation that supports unicast method invocation, the contents of this string includes transport specific information about the object (e.g., host name and port number) and an object identifier.

Objects are only cloneable using the Java language's default mechanism if they support the java.lang.Cloneable interface. Remote objects do not implement this interface, but do implement the clone method so that if subclasses need to implement Cloneable, the remote classes will function correctly.

Cloning a reference to a remote object is a local operation and cannot be used by clients to create a new remote object.

For RemoteServer objects, clone is implemented to make a new remote object distinct from the original. Cloning a remote object is only available in the server process where the remote object exists. If a remote object does not extend RemoteServer, it must implement its own version of clone and be able to export a cloned object.

The clone method for a remote object is defined to return a reference to the remote object. This operation does not copy any contents of the remote object, it simply returns a reference (since determining contents would require a remote call, and clone does not have RemoteException in its throws clause which would be raised in the event of a remote call failure).

The finalize method is used in specific circumstances depending on the type of remote object (for example, if a remote object is one that can be activated, some cleanup may be necessary).

There are several other methods defined in the class Object. These methods, however, are declared as final, which means that they cannot be overridden in an extended class. The methods are: getClass, notify, notifyAll, and wait.

The default implementation for getClass is appropriate for all Java objects, local or remote. The method needs no special implementation for remote objects. When used on a remote object, the getClass method reports the exact type of the generated stub object. Note that this type reflects only the remote interfaces implemented by the object, not its local interfaces.

The wait/notify methods of Object deal with waiting and notification in the context of Java's threading model. While use of these methods for remote objects does not break the Java threading model, these methods do not have the same semantics as they do for local Java objects. Use of these methods would only operate on the client's local reference to the remote object, not the actual object at the remote site. Since these methods are final, they cannot be extended to have behavior specific to remote objects.

Due to the differing failure modes of local and remote objects, distributed wait and notification requires a more sophisticated protocol between the entities involved (so that, for example, a client crash does not cause a remote object to be locked forever), and as such, cannot be easily fitted into the local threading model in Java. Hence, a client can use notify and wait methods on a remote reference, but that client must be aware that such actions will not involve the actual remote object, only the local proxy (stub) for the remote object.

Parameter Passing in Remote Invocation

A parameter of any Java type can be passed in a remote call. These types include both Java primitive types and Java objects (both remote and non-remote).

The parameter passing semantics for remote calls are the same as the Java semantics *except*:

- non-remote objects contained in a parameter of a remote call are passed by *copy*; and,
- non-remote objects returned as the result of a remote call are also passed by *copy*.

That is, when a non-remote object is passed in a remote call, the content of the non-remote object is copied before invoking the call on the remote object. Thus, there is no relationship between the non-remote object the client holds and the one it sends to a remote server in a call. For example, let's suppose that the remote object bank has a method to obtain the bank account given a name and social security number; the account information info is not a remote object but a local Java object:

```
Bank bank = ...;
String ssn = "999-999-9999";
AccountInfo info =
     new AccountInfo("Robin Smith", ssn);
BankAccount acct = bank.getAccount(info);
```

```
info.setName("Robyn Smith");
```

The contents of the object info is copied before invoking the remote call on the bank. A client can make changes to info without effecting the server's copy and vice versa.

Locating Remote Objects

A simple bootstrap name server is provided for storing named references to remote objects. A remote object reference can be stored using the URL-based interface java.rmi.Naming.

For a client to invoke a method on a remote object, that client must first obtain a reference to the object. A reference to a remote object is usually obtained as a return value in a method call. The RMI system provides a simple bootstrap name server from which to obtain remote objects on given hosts. The Naming interface provides Uniform Resource Locator (URL) based methods to lookup, bind, rebind, unbind and list the name and object pairings maintained on a particular host and port.

Here's an example of how to bind and lookup remote objects:

```
BankAccount acct = new BankAcctImpl();
URL url = new URL("rmi://zaphod/account");
// bind url to remote object
java.rmi.Naming.bind(url, acct);
// ...
// lookup account
acct = java.rmi.Naming.lookup(url);
```

In the current implementation, a "naming" registry contains a non-persistent database of name-object bindings. This database does not survive system crashes.

4 System Architecture

We have designed our RMI system in order to support the distributed object model discussed above. The system consists of three basic layers: the *stub/skeleton layer*, *remote reference layer*, and *transport*. A specific interface and protocol defines the boundary at each layer. Thus, each layer is independent of the next and can be replaced by an alternate implementation without effecting the other layers in the system. For example, the current transport implementation is TCP-based (using Java sockets), but a transport based on UDP could be used interchangeably.

To accomplish transparent transmission of objects from one address space to another, the technique of *pickling* [14] (designed specifically for Java) is used.

Another technique, that we call *dynamic stub loading*, is used to support client-side stubs which implement the same set of remote interfaces as a remote object itself. Since a stub of the exact type is available to the client of a remote object, a client can use Java's built-in operators for casting and typechecking remote interfaces.

Architectural Overview

The three layers of the RMI system consist of the following:

- stub/skeletons — client-side stubs (proxies) and server-side skeletons (dispatchers)
- remote reference layer — invocation behavior and reference semantics (e.g., unicast, multicast)
- transport — connection set up and management and remote object tracking

The application layer sits on top of the RMI system.

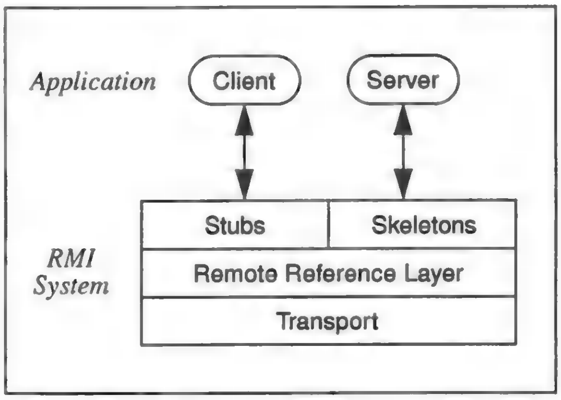


Figure 3. System Architecture

Figure 3 is an illustration of the layers of the RMI system. A remote method invocation from a client to a remote server object travels down through the layers of the RMI system to the client-side transport, then up through the server-side transport to the server. The rest of this section summarizes the functionality at each layer in the system.

A client invoking a method on a remote server object actually makes use of a *stub* or proxy for the remote object as a conduit to the remote object. A client-held reference to a remote object is a reference to a local stub. This stub is an implementation of the remote interfaces of the remote object and forwards invocation requests to that server object via the remote reference layer.

The *remote reference layer* is responsible for carrying out the semantics of the *type* of invocation. For example this layer is responsible for handling unicast or multicast invocation to a server. Each remote object implementation chooses its own invocation semantics—whether communication to the server is unicast, or the server is part of a multicast group (to accomplish server replication).

Also handled by the remote reference layer are the reference semantics for the server. For example, the remote reference layer handles live and/or persistent references to remote objects. Persistent object references are required in order to activate objects to support long-running servers.

The *transport* is responsible for connection set-up with remote locations and connection management, and also keeping track of and dispatching to remote objects (the targets of remote calls) residing in the transport's local address space.

In order to dispatch to a remote object, the server's transport forwards the remote call up to the remote reference layer (specific to the server). The remote reference layer handles any server-side behavior that needs to be done before handing off the request to the server-side skeleton. The skeleton for a remote object makes an up-call to the remote object implementation which carries out the actual method call.

The return value of a call is sent back through the skeleton, remote reference layer and transport on the server side, and then up through the transport, remote reference layer and stub on the client side.

Stub/Skeleton Layer

The stub/skeleton layer is the interface between the application layer and the rest of the RMI system. This layer does not deal with specifics of any transport, but transmits data to the remote reference layer via the abstraction of *marshal streams*. Marshal streams employ a mechanism called *pickling* which enables Java objects to be transmitted between address spaces. Objects transmitted using the pickling system are passed by copy to the remote address space.

A *stub* for a remote object is the client-side proxy for the remote object. Such a stub implements all the interfaces that are supported by the remote object implementation. A client-side stub is responsible for:

- initiating a call to the remote object (by calling the remote reference layer)
- marshaling arguments to a marshal stream (obtained from the remote reference layer)
- informing the remote reference layer that the call should be invoked
- unmarshaling the return value from a marshal stream
- informing the remote reference layer that the call is complete

A *skeleton* for a remote object is a server-side entity that contains a method which dispatches calls to the actual remote object implementation. The skeleton is responsible for:

- unmarshaling arguments from the marshal stream
- making the up-call to the actual remote object implementation
- marshaling the return value of the call onto the marshal stream

Remote Reference Layer

The remote reference layer deals with the lower level transport interface. This layer is also responsible for carrying out a specific invocation protocol which is independent of the client stubs and server skeletons.

Each remote object implementation chooses its own invocation protocol that operates on its behalf. Such an invocation protocol is fixed for the life of the object. Various invocation protocols can be carried out at this layer, for example:

- unicast invocation
- multicast invocation
- support for a specific replication strategy
- support for a persistent reference to the remote object (enabling activation of the remote object)
- reconnection strategies (if remote object becomes inaccessible)

These invocation protocols are not mutually exclusive, but may be combined. For example, a remote object may require both persistent reference semantics and replication. Both of these protocols would be carried out in the remote reference layer.

The invocation protocol is divided into two cooperating components: the client-side and the server-side components. The client-side component contains information specific to the remote server (or servers, if invocation is to a multicast group) and communicates via the transport to the server-side component. During each method invocation, the client and server-side components are given a chance to intervene in order to accomplish the specific invocation and reference semantics. For example, if a remote object is part of a multicast group, the client-side component can forward the invocation to the multicast group rather than just a single remote object.

In a corresponding manner, the server-side component is given a chance to intervene before delivering a remote method invocation to the skeleton. This component, for example, could handle ensuring atomic multicast delivery by communicating with other replicas in the multicast group.

The remote reference layer transmits data to the transport layer via the abstraction of a stream-oriented *connection.* The transport takes care of the implementation details of connections. Although connections present a streams-based interface, a connectionless transport may actually be implemented beneath the abstraction.

Transport

In general, the transport of the RMI system is responsible for:

- setting up connections to remote address spaces
- managing connections
- monitoring connection liveness
- listening for incoming calls
- maintaining a table of remote objects that reside in the local address space
- setting up a connection for an incoming call
- locating the dispatcher for the target of the remote call and passing the connection to this dispatcher

The concrete representation of a remote object reference consists of an endpoint and an object identifier. We call this representation a *live reference*. Thus, given a live reference for a remote object, a transport can use the endpoint to set up a communication channel to the address space in which the remote object resides. On the server side, the transport uses the object identifier to look up the target of the remote call.

The transport for the RMI system consists of four basic abstractions (based somewhat on the transport of the Modula-3 network object system):

- Endpoint — An endpoint denotes an address space. In the implementation, an endpoint can be mapped to its transport. That is, given an endpoint, a specific transport instance can be obtained.
- Transport — The transport abstraction manages channels. Each channel is a virtual connection between two address spaces. Within a transport, only one channel exists per pair of address spaces, the local address space and a remote address space. Given an endpoint to a remote address space, a transport sets up a channel to that address space. The transport abstraction is also responsible for accepting calls on incoming connections to the address space, setting up a connection object for the call, and dispatching to higher layers in the system.
- Channel — Abstractly, a channel is the conduit between two address spaces. As such, it is responsible for managing connections between the local address space and the remote address space for which it is a channel.
- Connection —A connection is the abstraction for transferring data (performing input/output).

A transport defines what the concrete representation of an endpoint is, so multiple transport implementations may exist. The design and implementation also allow multiple transports per address space (so both TCP and UDP can be supported in the same address space). Thus, client and server transports can negotiate to find a common transport between them.

Garbage Collection

In a distributed system, just as in the local system, it is desirable to automatically delete those remote objects that are no longer referenced by any client. This frees the programmer from needing to keep track of a remote object's clients so that the remote object can terminate appropriately. RMI uses a reference counting garbage collection algorithm similar to the one used for Modula-3 Network Objects [2].

To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each Java virtual machine. When a live reference enters a Java virtual machine its reference count is incremented. The first reference to an object sends a "referenced" message to the server for the object. As live references are found to be unreferenced in the local virtual machine, their finalization decrements the count. When the last reference has been discarded an unreferenced message is sent to the server. Many subtleties exist in the protocol, most related to maintaining the ordering of referenced and unreferenced messages to insure the object is not prematurely collected.

When a remote object is not referenced by any client, the RMI runtime refers to it using a weak reference. The weak reference allows the Java virtual machine's garbage collector to discard the object if no other local references to the object exist. The distributed garbage collection algorithm interacts with the local Java virtual machine's garbage collector in the usual ways by holding normal or weak references to objects. As in the normal object life-cycle, finalize will be called after the garbage collector determines that no more references to the object exist.

As long as a local or remote reference to a remote object exists, the remote object cannot be garbage collected and it may be passed in remote calls or returned to clients. Passing a remote object adds the client or server to which it was passed to the remote object's referenced set. A remote object needing unreferenced notification must implement the java.rmi.server.Unreferenced interface. When those references no longer exist, unreferenced will be invoked. unreferenced is called when the set of references is found to be empty so it may be called more than once. Remote objects are only collected when no more references, either local or remote, still exist.

Note that if there exists a network partition between a client and remote server object, it is possible that premature collection of the remote object will occur (since the transport may think that the client crashed). Because of the possibility of premature collection, remote references cannot guarantee referential integrity; in other words, it is always possible that a remote reference may in fact not refer to an existing object. An attempt to use such a reference will generate a RemoteException which must be handled by the application.

Dynamic Stub Loading

In remote procedure call systems, client-side stub code must be generated and linked into a client before a remote procedure call can be done. This code may be either statically linked into the client or linked in at run-time via dynamic linking with libraries available locally or over a network file system. In either the case of static or dynamic linking, the specific code to handle an RPC must be available to the client machine in compiled form.

This approach to code linking is static in that the stub code must be compiled and directly available to the client in binary-compatible form at compile time and at run time. Also with these systems, the stub code that the client uses is determined and fixed at compile time. Because of the static nature of the stub code available to clients in such systems, the code may not be the actual stub code that the client needs at run time, but perhaps the closest matched code that can be determined at compile time. For example in an RMI system, perhaps only a supertype (less specific form) of a more specific stub is available to the client at run-time. This code mismatch can lead to run-time errors if the client in fact needs a subtype (more specific form) of the stub that has been linked in at compile-time.

Our approach solves this code mismatch by loading the exact stub code (in Java's architecture neutral bytecode format) at run-time to handle method invocations on a remote object. This mechanism, called *dynamic stub loading*, exploits the Java mechanism for downloading code.

Dynamic stub loading is used only when code for a needed stub is not already available. The argument and return types specified in the remote interfaces are made available using the same mechanism. Loading arbitrary classes into clients or servers presents a potential security problem; this problem is addressed by requiring that a security manager check any classes downloaded for RMI.

In this scheme, client-side stub code is generated from the remote object implementation class, and therefore supports the same set of remote interfaces as supported by the remote implementation. Such stub code resides on the server's host (or perhaps another location), and can be downloaded to the client on demand (if the correct stub code is not already available to the client). Stub code for a remote implementation could be generated on-the-fly at the remote site and shipped to the client or could be generated on the client-side from the list of remote interfaces supported by the remote implementation.

Dynamic stub loading employs three mechanisms: a specialized Java class loader, a security manager, and the pickling system. When a remote object reference is passed as a parameter or as the result of a method call, the marshal stream that transmits the reference includes information indicating where the stub class for the remote object can be loaded from, if its URL is known.

A marshal stream is implemented by an underlying pickle stream. Pickle streams provide an opportunity to embed information for each class and object that is transmitted. When transmitting class information for a remote object being marshaled, a marshal stream embeds a URL that specifies where the stub code resides. Thus, when a reference to a remote object is unmarshaled at the destination site, the marshal stream can locate and load the stub code (via the specialized class loader, checked by the security manager) so that the correct stub is available to the client.

Security

The security of a process using RMI is protected by existing Java mechanisms of the security manager and class loader. The security manager regulates access to sensitive functions, and the class loader makes sure that loaded classes are subject to the security manager and adhere to the standard Java safety guarantees.

The JDK (Java Developer Kit) 1.0 security manager does not regulate resource consumption, so the current RMI system has no mechanisms available to prevent classes loaded from abusing resources. As new security manager mechanisms are developed to control resource use, RMI will use them.

The Applet Environment

In the applet environment, the AppletSecurityManager and AppletClassLoader are used exclusively. RMI uses only the established security manager and class loader. In this environment remote object stubs, parameter classes and return object classes can be loaded only from the applet host or its designated code base hosts. This requires that applet developers install the appropriate classes on the applet host.

The Server Environment

In the server environment, where a Java process is being used to serve RMI requests, the server may need to use a security manager to isolate itself from stub misbehavior. The server functions will usually be implemented by classes loaded from the local system and therefore not subject to the restrictions of the security manager. If the remote object interfaces allow objects, either local or remote, to be passed to the server, then those object classes must be accessible to the server. Usually those classes will be built-in classes or will be defined by the server. As long the classes are available locally there is no need for a specialized security manager or stub loader. To support this case, if no security manager is specified, stub loading from network sources is disabled.

When a server is passed a remote object for which it has no corresponding stub code, it may also be passed the location from which the classes for that remote object may be loaded. Two properties control if and from where the stub class can be loaded.

java.rmi.server.ClientClassDisable controls whether the URL's supplied by clients are used; if set to true, URL's from clients are ignored and stub classes are loaded using the stub class base.

java.rmi.server.StubClassBase defines the URL from which stub classes will be loaded. This is the URL that is passed along with remote object references so clients will know the location from which to load stub classes.

The StubClassLoader is a specialized class loader used by the RMI runtime to load classes. When loading any class, the runtime first attempts to use this class loader. If it succeeds those classes will be subject to the current security manager and any classes the stub needs will be loaded and then regulated by that security manager. If the security manager disallows creating the class loader, the class (including stub classes) will be loaded using the default Class.forName mechanism. Thus, a server may define its own policies via the security manager and stub loader and the RMI system will operate within them.

5 Related Work

The Common Object Request Broker Architecture (CORBA) [11] is designed to support remote method invocation between heterogeneous languages and systems. In CORBA, distributed objects are described in an interface definition language (IDL). IDL presents its own object model, and interfaces defined in IDL must be mapped into a target language and object model.

Because of IDL's language neutrality, the semantics of its object model does not match the object model semantics of any implementation language. This mismatch inhibits seamless integration of the CORBA distributed object model into any specific target language. Hence, programmers must deal with two very different object models when writing distributed programs: the local object model of the language, and the distributed object model mapped from IDL.

Our system differs from CORBA in two essential ways: it language-dependence and its ability to load stub code dynamically. Since our system is language-dependent, we can integrate the distributed object model more closely with the target language, Java. Also, systems that are CORBA compliant are unable to exploit the use of dynamic stub loading, since CORBA generally assumes that stub code is linked in at compile time.

Our approach is more akin to the Modula-3 (M3) network object system [3]. The Modula-3 system supports remote method invocation on objects in a language-dependent fashion (i.e., the system does not support interoperability with other languages). A second similarity is that the M3 system transmits objects via pickling. Our RMI system is similar in those respects (it depends on the architecture neutrality of Java bytecodes); however, our system is less static in its determination of stub code. The M3 network object system uses the closest matching stub code (called the most-specific surrogate) available at compile-time, rather than our approach in which the exact matching stub code is determined at run-time and downloaded over the network if such code is unavailable on the client.

The implementation of our system is similar to the M3 system in another respect: that is, the inclusion of a distinct abstraction for the transport. While the network object system has a similar notion of a transport abstraction, it does not include a separate remote reference layer to handle varying types of invocation semantics. Because of this limitation, this type of functionality is not easily layered on the network object system without adding some burden to the programmer.

Spring [7] is an object oriented operating system designed as a successor to UNIX. Spring has the notion of a subcontract [8] which has similar functionality to the remote reference layer in the RMI system. Our system differs in that the remote reference layer has a narrower interface that is more tailored to handling remote method invocation semantics. Subcontract is also very intimate with its doors-based transport, and as such does not support alternate transport implementations as readily as our approach.

Like CORBA, Spring uses an interface definition language to describe remote objects. Spring uses marshaling code generated from IDL descriptions of objects, whereas our system pickles exact representations of objects at run-time.

6 Future Work

The current system supports unicast remote method invocation to remote objects in Java. The system also implements pickling, dynamic stub loading, and garbage collection. We have fully designed and partially implemented activation for distributed objects in this framework. This effort is on-going. Also included will be the capability for server replication in this paradigm.

7 Conclusions

Our RMI system design leverages the two basic assumptions of platform homogeneity and language-dependence. We can assume homogeneity due to the architecture neutrality that the Java virtual machine provides. Since we are able to focus on language-dependent distributed objects, the resulting system presents a simple model that fits well into the Java framework, is highly flexible, and is accessible on a wide variety of machines.

Availability

The Java RMI system will be released with JDK 1.1. Early access versions of this system can be obtained from the http://java.sun.com web site.

Acknowledgments

We would like to thank Eduardo Pelegri-Llopart and Peter Kessler for useful discussions during the development of this system.

Authors

Ann Wollrath (ann.wollrath@sun.com) is a Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object-oriented programs.

Roger Riggs (roger.riggs@sun.com) is a staff engineer at Sun Microsystems Laboratories, East Coast Division, working on large scale distribution and reliable distributed systems. Prior to joining the group, he worked on client-server text retrieval and CORBA based information retrieval, and scabability issues on the Web.

Jim Waldo (jim.waldo@sun.com) is a Senior Staff Engineer at Sun Microsystems Laboratories, East Coast Division, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

References

[1] Arnold, Ken, and James Gosling, *The Java™ Programming Language*. Addison-Wesley (1996).

[2] Birrell, Andrew, David Evers, Greg Nelson, Susan Owicki, and Edward Wobber, *Distributed Garbage Collection for Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 116 (1993).

[3] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, *Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[4] Ellis, Margaret A., and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Addison-Wesley (1990).

[5] Goldberg, Adele, and David Robson, *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley (1983).

[6] Gosling, James, Bill Joy, and Guy Steele, *The Java™ Language Specification*. Addison-Wesley (1996).

[7] Hamilton, G., and P. Kougiouris, "The Spring Nucleus: A Microkernel for Objects." *USENIX Summer Conference* (July 1993).

[8] Hamilton, Graham, Michael L. Powell, and James G. Mitchell, *Subcontract: A Flexible Base for Distributed Programming*. Sun Microsystems Laboratories Technical Report, SMLI TR-93-13 (April 1993).

[9] Mullender, Sape (ed.), *Distributed Systems* (second edition). Addison-Wesley (1993).

[10] Nelson, Greg (ed.), *Systems Programming with Modula-3*. Prentice Hall (1991).

[11] The Object Management Group, *Common Object Request Broker: Architecture and Specification*. OMG Document Number 91.12.1 (1991).

[12] Pinson, Lewis J. and Richard S. Wiener, *Objective C: Object-Oriented Programming Techniques*. Addison-Wesley (1991).

[13] Rago, Steven A., *UNIX System V Network Programming*. Addison-Wesley (1993).

[14] Riggs, Roger, Jim Waldo, Ann Wollrath, and Krishna Bharat, "Pickling State in the Java™ System." The *2nd USENIX Conference on Object-Oriented Technologies*. (1996).

[15] Waldo, Jim, Geoff Wyant, Ann Wollrath, and Sam Kendall, *A Note on Distributed Computing*. Sun Microsystems Laboratories Technical Report, SMLI TR-94-29 (November 1994).

[16] Wollrath, Ann, Geoff Wyant, and Jim Waldo, "Simple Activation for Distributed Objects." *1st USENIX Conference on Object-Oriented Technologies*. Monterey, CA (June 1995), pp. 1-11.

Smart Messages: An Object-Oriented Communication Mechanism for Parallel Systems

Eshrat Arjomandi
Dept. of Computer Science
York University
Toronto, Ontario
eshrat@cs.yorku.ca

William G. O'Farrell & Gregory V. Wilson
Center for Advanced Studies
IBM Canada Ltd.
Toronto, Ontario
{billo,gvwilson}@vnet.ibm.com

Abstract

ABC++ is a portable object-oriented type-safe class library for parallel programming in C++. It supports active objects, synchronous and asynchronous object interactions, and object-based shared regions on both shared- and distributed-memory parallel computers. ABC++ is written in, and compatible with, standard C++: no language extensions or pre-processors are used. This paper focuses on its use of an object-oriented technique called *smart messages* to support object interactions. Smart messages demonstrate the effectiveness of object-oriented programming in encapsulating low-level details of concurrency and in improving software portability.

1 Introduction

While massively-parallel computers and networks of workstations (NOWs) are now widely available, good programming systems for them are not. Rapid architectural change has meant that the levels of code re-use which are taken for granted in conventional computing environments are still only dreamt of by parallel programmers. Many groups are now trying to insulate users from such changes using the abstraction and polymorphism facilities of the object-oriented programming (OOP) paradigm.

Language designers have three options in integrating OOP and concurrency: create a new language, add new features to an existing language, or construct libraries to work with an existing language. The first two approaches give much more freedom to experiment with new ideas, but experience shows that application programmers are very reluctant to translate existing codes, or develop new ones, to make use of a non-standard environment. It is also very difficult for any but the largest of research groups to develop the "boring" (i.e. sequential) portion of a non-standard language system, and keep its capabilities and performance competitive with that of standard systems. A library-based approach, on the other hand, must find a way to accommodate language features which were designed (or which "just happened") long before parallelism was an issue.

C++ is rapidly replacing C as the language of choice for systems programming, and is starting to be adopted by scientific programmers, who have traditionally been the largest users of parallel computers. For this reason, numerous attempts have been made to add concurrency to C++ [2, 4, 5, 6, 7, 8, 9, 10, 12]. Most such systems require extensive compiler extensions and/or preprocessors. Attempts using purely a class library approach have often not fully utilized OOP in their design and implementation, or have imposed unreasonable limitations on the users. For example, some libraries [2, 7, 8] limit the height of the user's class hierarchy to one, require explicit use of *wait* and *alert* routines for synchronization, or require explicit manipulation of message queues to manage object interaction.

ABC++ is a class library to support parallel programming in C++. Its initial implementation [1] demonstrated that many of the limitations listed above can be eliminated without resorting to language extensions or pre-processors. ABC++'s concurrency model supports *active objects*—objects which have their own threads of control, and which can run simultaneously with other active objects—and remote method invocation for inter-object communication. However, the first version of ABC++ had several limitations, such as compiler and architecture dependencies and a lack of type-safety checks.

One of the difficulties in the design and implementation of a concurrent class library for C++ is constructing a flexible, architecture-independent communication mechanism for object interaction. For example, while the implementation of method invocation is straightforward in a sequential environment, it can be problematic on distributed-memory machines. For the sake of efficiency and simplicity, most message-passing systems require the data to be sent to occupy a contiguous block of memory. Since programmers often want to pass logically-unconnected values as parameters to a single method invocation, some way to *marshal* these values into a contiguous block of bytes must be found. Furthermore, the type of the class whose method is being invoked, and the name of the invoked method, must somehow be communicated along with these parameters.

The implementation of object interaction in ABC++ uses the data abstraction, genericity, and polymorphism facilities of C++ to solve these problems. Based on *active messages* [14], we introduce an object-oriented communication mechanism, *smart messages*, that captures the required data and typing information. Active messages is an asynchronous communication model designed to minimize network latency by allowing communication and computation to be overlapped. In this model, a process sends a message to another remote process which contains not only data, but also the address of a user-defined function, called a *handler*. When the message arrives, this handler is invoked. Its duty is to extract the message from the network and integrate it with the ongoing computation.

A smart message is an object which carries not only data, but also the typing information and all the other functionality provided by its class. This paper shows how smart messages are used and implemented in ABC++ to provide a portable, object-oriented solution to active object creation and interaction. Section 2 provides a brief overview of ABC++. Section 3 discusses the implementation of smart messages. Active object creation and interactions are covered in Sections 4 and 5 respectively. An appendix discusses issues related to automatic type conversion during method invocation and object creation.

2 A Brief Overview of ABC++

This section provides a brief introduction to major components of ABC++. For more detail, see [13].

ABC++ is a class library written in standard C++ to support parallel programming in C++. Its run-time system provides portability across uniprocessors, shared-memory multiprocessors, homogeneous NOWs, and massively parallel machines. It is presently available for IBM RISC System/6000 workstation networks and SP supercomputers.

ABC++ supports concurrency through *active objects*. An active object possesses its own thread of control and can be created on any processor. To allow active instances of a class to be created, the class must publically inherit from class `Pabc` provided in the header file `ABC++.h`.

The thread of an active object executes method `main()`. `Pabc` provides a default `main()`, which repeatedly accepts method invocations from other active objects; classes derived from `Pabc` may redefine `main()`. An active object terminates only when the whole program terminates.

An active object can control which of its methods are invocable using the functions `Paccept` and `Paccept_any`. The arguments to `Paccept` are the names of methods that the object is willing to serve. The use of `Paccept_any` signals that the object is willing to serve any of its public methods. Each call to `Paccept` or `Paccept_any` matches exactly one method invocation. The present implementation of ABC++ only allows a call to `Paccept` or `Paccept_any` from within the `main()` routine of active objects. A call to these functions elsewhere causes an exception to be thrown.

An active object is created in two steps. First, a handle is declared using the template `Pabc_pointer`. This handle is then used to reference the active object.

```
class C_Active : public Pabc {
  body of active class
};
...
Pabc_pointer<C_Active> active_p;
```

Next, the function `Pabc_create` is called to create an active object and bind a reference to it to the handle. `Pabc_create` is in fact a family of overloaded function templates with varying numbers of arguments. The first (optional) argument to `Pabc_create` allows users to specify a particular processor on which the active object should be created. If this argument is not provided, by default, the ABC++ run-time system determines where to create the new active object. The next argument is a handle for the active object being created. The remaining zero or more arguments are arguments to the active object class constructor. For example, the following code creates an active instance of `C_Active`. The value `12345` is passed to `C_Active`'s constructor.

```
Pabc_create(active_p, 12345);
```

Active objects in ABC++ communicate through remote method invocation (RMI). Both synchronous and asynchronous RMIs are supported. ABC++'s template functions Pvalue and Pvoid perform blocking invocations of methods returning or not returning a value, respectively. For example, the following line of code invokes the method foo of the active object bound to active_p, passing the integer 456 as an argument:

```
int i = Pvalue(active_p, C_Active::foo, 456);
```

ABC++ requires the specification of fully qualified method names (as in C_Active::foo) as an argument to Pvalue and other template functions used in active object communication. This is sometimes tedious; however, since this argument is a function pointer, there is no interference with the virtual function mechanism. In particular, if active_p is declared as a base class pointer but is pointing to an object of a derived class, the foo method of the "right" class will be invoked, even if the Pvalue call is made using a base class function pointer.

Ppar_void and Ppar_value implement asynchronous RMI. Their arguments are identical to those taken by Pvoid and Pvalue. However, neither Ppar_void nor Ppar_value block the caller. Instead, the calling object proceeds with its activity as soon as the arguments to the call have been copied to a safe place. Ppar_void is used for asynchronous invocation of void methods; Ppar_value is used when a result is expected.

Futures [11, 6, 12] are used to receive results of asynchronous RMIs. A future is an instance of the Pfuture class template. When Ppar_value is called, the future is marked as pending. The future is resolved when a value becomes available for it; any attempt to read its value before it is resolved blocks its reader. The following code shows how ABC++ futures are used:

```
Pfuture<int> iF;
iF = Ppar_value(active_p, C_Active::foo, 456);
int i = iF; // block until result becomes available
```

The mechanisms presented so far allow for direct communication among active objects. ABC++ supports a second model of communication and synchronization which allows for indirect object interactions. *Parametric shared regions*, or PSRs, are used to provide the illusion of shared memory. A PSR may be any C++ object. ABC++'s run-time system provides copies of PSRs to other processors when and as they are needed. Consistency of shared regions are guaranteed by ABC++'s run-time system. Detailed discussions of how PSRs are used and implemented are outside the scope of this paper. For more detail on PSRs see [13].

3 Smart Messages

In order to support the model described in the previous section, ABC++'s run-time system must provide a flexible, architecture-independent communication mechanism. For example, a message requesting object creation must carry any constructor arguments required to create that object, the request to call new, and the type of the class whose constructor must be called. The arguments to the constructor must be automatically marshalled into a contiguous block of memory. Similarly, a remote method invocation (RMI) must contain some identification of the object whose method is being invoked, the method itself, and any arguments that method requires.

ABC++ utilizes data abstraction, genericity[1], and polymorphism to create a unified framework called *smart messages* to handle both remote object creation and remote method invocation. A smart message is an instance of a *smart class*. The instance variables of each particular smart class contain the information required to carry out the desired operation. A designated method of the smart class, called do, encapsulates the requested operation. For example, if the request is for object creation, ABC++ will automatically create a smart class by template expansion. This class's instance variables will have the same types as its constructor arguments; upon instantiation, its instance variables will be initialized by copying the user-supplied constructor arguments. This class's do method will be defined to invoke new using its instance variables. When an instance of this class is received at a remote processor, that processor will invoke its do method to create an object of the required class.

Since a request for object creation or method invocation may involve a varying number of arguments, ABC++ provides a family of smart classes with varying number of instance variables. Each of these classes has its own unique name. However, upon the arrival of a smart message at a destination processor, ABC++ must be able to invoke its do method. Polymorphism is used to allow this. All smart classes inherit from an abstract base class SmartMsg which provides a deferred method (called a *pure virtual function* in C++):

```
class SmartMsg {
 public:
  virtual void do() = 0;
};
```

Descendents of this class define do to perform either object creation or method invocation. The remainder

[1]Provided in C++ by templates.

of this section deals with smart classes implementing remote method invocation; Section 4 covers how smart messages implement remote object creation.

3.1 Marshalling Data

As mentioned earlier, ABC++ must marshal the arguments to remote operations. Smart messages allow this to be done in a type-safe way. Every smart message used for an RMI has at least two data members: a pointer to the method to be invoked, and a pointer to the object which is to execute the method. This pointer is defined in the address space of the processor on which that object executes; as will be seen in Section 4, such a pointer is embedded in an active object handle by `Pabc_create`. The remaining instance variables store the arguments to be passed to the invoked method. The do method of the smart message invokes the desired method using the values stored in the instance variables.

Smart messages should be able to encode the invocations of methods with varying number of parameters. This is handled by defining a family of class templates. The present implementation of ABC++ provides class templates with zero to 16 parameters. The following segment of code demonstrates the 1-parameter version for a method returning a value.

```
class C_Obj
{
  ···user-defined object class···
};

template<class C_Obj, class C_Ret, class C_Arg1>
class SmartMsg1: public SmartMsg
{
 public :
  C_Obj * obj_p;
  C_Ret (C_Obj::*meth_p)(C_Arg1);
  C_Arg1 arg1;

  SmartMsg1(Pabc_pointer<C_Obj> objHndl,
            C_Ret (C_Obj::*method)(C_Arg1),
            const C_Arg1 & a1)
  : obj_p(objHndl.object),
    meth_p(method),
    arg1(a1)
  {}

  void do()
  {
    C_Ret r = (obj_p->*meth_p)(arg1);
  }
};
```

The data member `obj_p` caches the object pointer from the active object handle given as a constructor argument. The data member `meth_p` caches a pointer to the method being invoked, while `arg1` stores a copy of the argument to be passed to that invocation. `SmartMsg1`'s do method takes a particular object as an argument, and calls the specified method on that object with the actual argument. This method can only be called safely in the address space of the processor on which the specified active object is running.

The most important aspect of `SmartMsg1` is its role in encapsulating the necessary information for invoking a method into a single object, so that the data is guaranteed to be contiguous in memory. Copying `sizeof(SmartMsg1)` bytes from `&sm1` (where `sm1` is a particular instance of the smart message class) is therefore guaranteed to copy the whole of the smart message.

3.2 Creating Smart Messages

Smart message classes are used by a series of overloaded function templates to create smart messages. The following function demonstrates how smart messages are created from the `SmartMsg1` class:

```
template<class C_Obj, class C_Ret, class C_Arg1>
C_Ret Pvalue(Pabc_pointer<C_Obj> & objHndl,
             C_Ret (C_Obj::*method)(C_Arg1),
             const C_Arg1 & a1)
{
  // create smart message
  SmartMsg1<C_Obj, C_Ret, C_Arg1>
    smartMsg(objHndl, method, a1);

  ···send the smart message to the desired destination···

  C_Ret r; // temporary storage for result
  ···assign value returned to r···
  return r;
}
```

The first statement in this function creates a smart message. This message is then sent to another processor and the calling object blocks until the result is returned. The mechanism used for sending the message is discussed in Section 4.

3.3 Remote Invocation

ABC++ provides smart classes in order to accommodate the invocation of methods taking up to 16 parameters. Therefore on the receiving side, a correct handle must be used to invoke method do. This is achieved with the help of polymorphism. As mentioned earlier, ABC++ provides an abstract base class, `SmartMsg`, with a single pure virtual function do. All smart classes publically inherit from this class. By obtaining a handle to SmartMsg on the receiving side, the virtual function mechanism of C++ will guarantee the invocation of the "right" do method.

4 Active Object Creation

Section 2 presented a brief introduction to how users can create active objects. In this section we show how smart messages are used in the implementation of remote active object creation.

As mentioned earlier, the function Pabc_create is called to create active objects. Pabc_create is a family of overloaded function templates with varying number of arguments. The first argument is a handle to the active object being created. The remaining zero or more arguments are arguments to the active object class constructor. A skeletal implementation of Pabc_create for active object constructors taking one argument is shown below:

```
template<class C_Obj, class C_Arg1>
void Pabc_create(Pabc_pointer<C_Obj> & objHndl,
                 const C_Arg1 & a1)
{
  ···check for pointer arguments···
  Proc p = procSet.select();
  SmartMsg_Create1<C_Obj, C_Arg1> smartMsg(a1);
  P__send(p, &smartMsg, sizeof(smartMsg));
  P__createReply<C_Obj> reply;
  P__recv(&reply, sizeof(reply));
  P__abcSetPtr(objHndl, reply.data, p);
}
```

The two template arguments to this function are the class of the active object being created, and the class of the object's constructor's argument. The function's formal parameter is the argument to use when constructing the active object. SmartMsg_Create1 is the smart class version for constructors requiring one argument. Versions of this function for constructors taking up to 16 arguments are provided, as are versions which allow a processor to be selected by the user, rather than by using the automatic load-balancing method of the procSet class.

After some (sequential) code to force the compiler to check that the constructor argument is not a pointer, Pabc_create creates a smart message containing the constructor argument. The data transfer function P__send is then called to ship the smart message to a daemon on the designated processor. The actual transport mechanism may be TCP/IP, MPI, or whatever else is convenient.

Next, Pabc_create creates a reply buffer, which will be used to store a pointer to the object generated on the remote processor, and then blocks its caller until a reply is received. Since inter-object communication can only be done through handles, this ensures that an active object never tries to send a request to another object which has not yet completed its own initialization. The value in this reply is a pointer to the active object which has just been created. The final line of Pabc_create extracts this pointer, and stores it in its handle argument. This pointer can then be extracted in subsequent RMIs.

Section 3.1 showed how smart classes marshal the arguments to a RMI in a type-safe manner. Smart classes for remote object creation are very similar to smart classes presented in Section 3.1 for RMI. These classes also inherit from the abstract base class SmartMsg. The instance variables (if any) of these smart classes store the constructor arguments needed to create the active object. The definition of the do() method simply calls new using its instance variables as the constructor arguments. The following code demonstrates the 1-argument version:

```
template<class C_Obj, class C_Arg1>
class SmartMsg_Create1: public SmartMsg
{
 public:
  Proc srcProc;
  C_Arg1 arg1;
  SmartMsg_Create1(const C_Arg1 & a1) :
    arg1(a1)
  {
    srcProc = procSet.this();
  }
  void do()
  {
    P__reply<C_Obj> reply;
    // create active object
    reply.data = new C_Obj(a1);
    P__send(srcProc, &reply, sizeof(reply));
  }
};
```

5 Data-Based Synchronization Using Futures

As stated in Section 2, ABC++ uses *futures* to implement data-based synchronization of remote method invocations. Futures are implemented by defining a template class to hold the results of asynchronous remote method invocations which return values. Each instance of the class Pfuture holds a reference to the future return value of an asynchronous invocation. Type conversion from the future type to the base type blocks until the return value actually arrives. A future can also be initialized with an actual object, in which case it acts just like a holder for that value without blocking. Assignment of futures to each other is well-defined. A partial signature of Pfuture is:

```
template<class T>
class Pfuture
{
 private:
  P__future_result<T> * future_result;

 public:
  // create futures
  Pfuture();
  Pfuture(const Pfuture<T>& fut);
  Pfuture(const T& value);
  // alias and overwrite futures
  Pfuture<T>& operator=(const Pfuture<T>& fut);
  Pfuture<T>& operator=(const T& value);
  // delete futures
  ~Pfuture();
  // convert to base type
  operator T() const;
  // test for completion
  int resolved() const;
};
```

The class P__future_result contains data and member functions to handle the future-resolution protocol. The behavior of instances of this class is independent of the data type encapsulated in any particular future. The methods of Pfuture itself allow futures to be created, assigned values, converted to their base type, and tested for completion.

Given this structure, the template function Ppar_value performs an asynchronous remote method invocation. When invoked, it marks its future argument as unresolved. Subsequent attempts to access the future's data will block until the future becomes resolved. As with Pvalue, a smart message is created to carry argument values and a method function pointer to the remote method whose operation is being invoked. The following segment of code illustrates the major components of Ppar_value. SmartMsgFuture1 is the 1-argument version of smart classes defined for future interactions.

```
template<class C_Obj, class C_Ret, class C_Arg1>
Pfuture<C_Ret>
Ppar_value(Pabc_pointer<C_Obj> objHndl,
           C_Ret (C_Obj::*method)(C_Arg1),
           const C_Arg1 & a1)
{
  ···check validity of arguments···

  // create future token for later reference
  Pfuture<C_Ret> token;

  // create smart message
  SmartMsgFuture1<C_Obj, C_Ret, C_Arg1>
    smartMsg(token, objHndl, method, a1);

  ···send message···

  return token;
}
```

The future object, token, created inside this function is a placeholder containing synchronization control information. It is returned to the caller so that references to it may block or complete, and a pointer to it embedded in the smart message so that the reply from the remote method invocation will have a way to identify its future.

6 Conclusion

This paper focussed on *smart messages*, an object-oriented technique in support of active object interactions in ABC++. We showed how to utilize the data abstraction, genericity, and polymorphism facilities of the object-oriented paradigm to create a unified framework to handle both remote object creation and remote method invocation.

ABC++ provides several commonly-used abstractions of parallelism within standard C++. It supports type-safe parameter marshalling, remote method invocation, and object-sized distributed shared memory without any pre-processors, post-processors, or language extensions. As a result, ABC++ is very portable, and does not require users to commit themselves to non-standard or poorly-supported tools. Future directions for ABC++ may include the incorporation of multiple consistency protocols as exemplified in Munin [3], and support for group operations in the form of data parallelism, or its higher-level counterpart, method parallelism.

Acknowledgments

ABC++ was developed at the Center for Advanced Studies at IBM Canada's Toronto Laboratory, in collaboration with researchers from several universities, including York, Syracuse, Toronto, and McGill. We wish to thank Frank Eigler for his work on implementing the current version of ABC++, and Howard Operowsky for his support and suggestions. We also wish to thank Tim Brecht of York University for his suggestions toward making the implementation of ABC++ more portable. Finally we thank the many people who contributed to the prototyping and testing of ABC++, including Young-il Choo, Jagdeep Dhillon, Ali Ghobadpour, Stephen Howard, Ivan Kalas, Gita Koblents, Henry Lee, Peter Milley, Fernando Nasser, S. David Pullara, Ilene Seelemann, and Susan Sim.

Appendix: Checking and Converting Argument Types

The templates shown in this paper are unnecessarily restrictive, as they do not allow type conversion during invocation. For example, because the types of both the formal argument to the method in `Pvalue`, and the actual argument given as `Pvalue`'s third parameter, are specified as `C_Arg1`, an invocation such as:

```
Pvalue(tHndl, TestClass::methodTakingInt, 'a');
```

would fail with a type-matching error.

ABC++ circumvents this with a slightly more convoluted definition of `Pvalue`:

```
template<class C_Obj, class C_Ret,
        class C_Formal1, class C_Actual1>
C_Ret Pvalue(Pabc_pointer<C_Obj> & objHndl,
             C_Ret (C_Obj::*meth)(C_Formal1),
             const C_Actual1 & a1)
{
  // check legality of type conversion
  C_Actual1 * actual_p = NULL;
  C_Formal1 * formal_p = actual_p;

  ···check for pointer arguments···

  ···rest of body as before···
}
```

This version of `Pvalue` takes three class parameters: the object's class, the type of the formal argument to the method being invoked, and the type of the actual argument being passed to the invocation. The first line of `Pvalue` creates a `NULL` pointer of the actual argument type; the second line then tries to assign from this pointer to a pointer of the formal type. If the actual type cannot be converted to the formal type, this conversion will fail. A modern optimizing C++ compiler, such as IBM's CSET++, can determine that these values are not subsequently used, and delete them during optimization. This technique therefore has no run-time cost.

The next statements in this modified `Pvalue` checks to ensure that the actual argument being passed is not a pointer. ABC++ does not (presently) allow pointer arguments to be passed during remote method invocation because there is no guarantee that the thing pointed to at the receiving end will bear any resemblance to the thing pointed to at the sending end. ABC++ forces the compiler to check for pointer parameters by providing templates that will match all pointer and non-constant reference arguments, plus one that will match immediate and constant reference arguments. We begin by noting that if the declared type of a formal argument is `const X&` (for some type `X`), then `C_Actual1` will be bound to the whole of `const X&` and not just to `X`. If the user attempts to pass a pointer or reference argument to a remote method invocation, the compiler will find two ways to unify this attempt with these templates. Since this ambiguity is illegal, the compiler will generate an error message using the mock argument name `no_pointer_argument_allowed`, and fail. These templates are:

```
template<class T>
void _P_ptr_invalid(
  T* no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  T* const no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T* no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T* const no_ptr_arg_allowed
) {}

template<class T>
void _P_ptr_invalid(
  const T& no_ptr_arg_allowed
) {}
```

For example, if a program attempts to pass an `int*` as a parameter to a remote method invocation, then during compilation, the compiler will unify `int` with `T` in the first template, but also unify `int*` with `T` in the last template (trying to create a function with a `const int * &` argument).

References

[1] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F.C. Eigler, and G. Gao, "ABC++:

Concurrency by Inheritance in C++", *IBM Systems Journal*, Vol. 34, No. 1, (1995), pp. 120-136.

[2] AT&T C++ Language System Release 2.0: Product Reference Manual, Select Code 307-146, AT&T Bell Laboratories, Murry Hill, NJ 07974 (1989).

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence", in *Proceedings of the 1990 Conference on Principles and Practice of Parallel Programming*, ACM Press, 1990.

[4] B. Bershad, E. D. Lazowska and H. M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming", *Software–Practice and Experience*, Vol. 18(8), (August 1988) pp. 713-732.

[5] P.A. Buhr, G. Ditchfield, R. A. Stroobosscher, B. M. Younger, and C. R. Zarnke, "μC^{++}: Concurrency in the Object-Oriented Language C++", *Software–Practice and Experience*, 22(2), (1992), pp. 137-172.

[6] R. Chandra, A. Gupta and J. Hennessy, "COOL: a Language for Parallel Programming", in *Languages and Compilers for Parallel Computing*, edited by D. Gelernter, A. Nicolau, D. Padua, MIT Press, (1990).

[7] T.W. Doeppner Jr., and Alan J. Gebele, "C++ on a parallel machine", Report CS-87-26, Department of Computer Science, Brown University, (November 1987).

[8] P. Gautron, "Porting and Extending the C++ Task System with the Support of Lightweight Processes", *USENIX* C++ *Conference Proceedings*, (1991), pp. 135-146.

[9] N.H. Gehani and W.D. Roome, "Concurrent C++: Concurrent Programming with Class(es)", *Software–Practice and Experience*, Vol. 18(12), (1988), 1157-1177.

[10] D. Grunwald, A User's Guide to AWESIME: An Object-Oriented Parallel Programming and Simulation System, Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder (1991).

[11] R. Halstead, "Multilisp: A Language for Concurrent Symbolic Computation", *ACM Transactions on Programming Languages and Systems*, October 1985.

[12] D. Kafura and K.H. Lee, "ACT++: Building a Concurrent C++ with Actors", *Journal of Object-Oriented Programming*, Vol. 3, No. 1, (1990), pp. 25-37.

[13] W. G. O'Farrell, F. Ch. Eigler, I. Kalas, and G.V. Wilson, ABC++ User Guide, "An Introduction to the IBM Parallel Class Library for C++", ABC++ Version 1, Release 1, IBM Canada, abc++@vnet.ibm.com, (1995).

[14] T. von Eicken, D.E. Culler, S.C. Goldstein, K.E. Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation", *Proceedings of the 19th International Symposium on Computer Architecture*, ACM Press, Gold Coast Australia, (May 1992).

Pickling State in the Java™ System

Roger Riggs, Jim Waldo, Ann Wollrath, Krishna Bharat
JavaSoft
{Roger.Riggs, Jim.Waldo, Ann.Wollrath}@Sun.COM, kb@cc.gatech.edu

Abstract

The Java™[1] system (hereafter referred to simply as "Java") inherently supports the transmission of stateless computation in the form of object classes. In this paper we address the related task of capturing the state of a Java object in a serialized form for the purposes of transmission or storage, to be used later in reconstituting an equivalent object. This is accomplished by a mechanism known as *pickling*[2][3][5].

Pickling is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. Unpickling is the complementary process of recreating objects from the serialized representation.

Pickling and unpickling extract from the Java Virtual machine, at runtime, any meta information needed to pickle the fields of objects. Class specific methods are only required to customize the pickling process.

1 Introduction

The Java™[1][4] system supports the transmission of stateless computation in the form of object classes. These can be dynamically loaded into a Java Virtual Machine, for immediate linking and execution. This enables a computation to be "cold-started" at new hosts, always starting with the same, initial state and at the same point in the computation. However, if we need to resume the computation with the state it had prior to transmission, or need to communicate state between cooperating processes, we would need to transmit objects as well.

1. Java and other Java-based names and logos are trademarks of Sun Microsystems, Inc., and refer to Sun's family of Java-branded products and services.

Pickling is the process of creating a serialized representation of objects. Pickling defines the serialized form to include the meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated.

Unpickling is the complementary process of recreating objects from the serialized representation. The meta information needed for pickling and unpickling is extracted directly from the Java Virtual machine at runtime. Class specific methods are only required to customize the pickling process.

There are many applications for pickling, including:

- Checkpointing of application state; support for persistent objects.
- Marshalling of objects as arguments to and returns from remote-method invocations.
- Serializing objects or object graphs so that they may be stored and retrieved as blobs in databases.

Goals for Pickling

We identify the following goals for pickling:

- Build a simple yet extensible mechanism for serializing and deserializing Java objects.
- Maintain the Java language's object semantics and extend the type and safety properties guaranteed by the Java language and type system to the serialized form.
- Be extensible to support, for example, marshalling and unmarshaling needed for remote object systems.
- Provide simple persistence of Java objects and be extensible to allow more sophisticated Java object storage.

In this paper we describe pickling in a Java system. Section 2 describes the Java Object Model. Section 3 describes the implementation. Section 4 describes type fingerprinting and Section 5 the structure of pickle streams. Section 6 looks at the integrity of sensitive data. Section 7 describes future work.

2 The Java Object Model

The pickling of Java objects is based on the availability at runtime of descriptions of Java objects. The Java Virtual Machine retains descriptions of Java object classes for its own use in loading and verifying classes. Pickling uses this information about objects to save and restore the state of objects.

The Java language is described in full elsewhere[4] so only those aspects relevant to saving an object's state will be described here. The Java language is a strongly typed object-oriented language with a syntax similar to C.

Java classes inherit implementations from at most one other class. All classes extend the base class java.lang.Object. Base classes are extended to define subclasses and may add methods and fields.

The Java language defines an interface class as an abstract class declaration that has no implementation. It contains only methods and constants. Java classes may implement multiple interfaces. To implement an interface the class must implement all of the methods specified by the interface. Pickling is only concerned with an object's state so the presence or absence of interfaces only contributes to the identification of the class, not it storage.

Each class may define zero or more methods and fields. Methods are procedures that apply to the class. There are two kinds of methods, those that operate on a specific object and class methods that do not operate on an instance when they are invoked.

The fields of a class hold the state of the class. Each field is strongly typed as either a reference to an object or to one of the built-in primitives types including integer, floating-point, character, and booleans. In the Java system strings and arrays are object types and have support in the language. Fields can be of two kinds, class and instance. Class fields are shared among all instances, instance fields are stored for each object.

Access to the fields of an object is specified as public, package, protected, or private. Public fields are accessible via any reference to the object. By default, the fields of an object are accessible to any object within the package of the referenced object. Protected fields are accessible to subclasses of the class that define them as well as to other classes in the same package. Private fields are only accessible to methods of the object.

The only mechanism to manipulate objects is by object references which are strongly typed. All non-primitive fields of an object refer explicitly to some object class, so the object is known to implement that classes behavior. A reference to an object may be null, signifying that it does not refer to any object.

When a class is extended the subclass is given access to the superclass's public, package, and protected fields, but not the private fields. Subclasses may use all of the accessible fields of their superclasses.

The complete state of an object is held in its class and instance fields and in the class and instance fields of all of its superclasses. Since the class fields are shared among all instances, difficulties arise in transporting and restoring their contents. It is the state of the instance fields that must be written and read from a stream in order to reconstruct the complete object.

3 Simplified Pickling

We present a simple and flexible approach to picklingJava objects. The approach extends the support provided by the Java system for primitive data types. The DataOutput and DataInput interfaces define methods for writing and reading integer types, floating point types, booleans, characters, strings, and arrays of bytes. The DataOutputStream and DataInputStream classes implement these interfaces and perform the encoding and serialization of the primitive types.

For pickling, the ObjectOutput and ObjectInput interfaces extend DataOutput and DataInput interfaces respectively to include writeObject and readObject methods. The classes ObjectOutputStream and ObjectInputStream extend DataOutputStream and DataInputStream respectively to implement pickling of all object types. Special handling is required for Strings, arrays, and Class objects.

An example use of pickling and unpickling is shown in Figure 1.

```
// Pickle today's date to a file.
FileOutputStream os =
                new FileOutputStream("tmp");
ObjectOutputStream p = new ObjectOutputStream(os);

p.writeObject("Today");
p.writeObject(new Date());
p.flush();

// Unpickle a string and date from a file.
FileInputStream in = new FileInputStream("tmp");
ObjectInputStream p = new ObjectInputStream(in);

String today = (String)p.readObject();
Date date = (Date)p.readObject();
```

Figure 1: Pickling and Unpickling Example

The application sets up an output stream and writes a sequence of objects or primitives types. Pickling serializes the objects and objects reachable from them. The objects are written to a stream along with type information so that they can be reinstantiated as equivalent objects by the complementary, *unpickling* mechanism. Shared references within the set of objects are preserved and complex data-graphs are restored with the same structure.

Pickling Interfaces

Pickling utilizes three interfaces:

- The client interface drives the pickling process. It is called with individual objects or primitives that are to be pickled.
- The specials interface allows methods of the object to implement the serialization and deserialization for its own fields.
- The subclass interface allows pickling to be extended to allow additional information about classes and objects to be written to and read from the stream.

Pickling Client Interface

The client interface consists of ObjectOutput and ObjectInput interfaces which are extensions of the basic DataOutput and DataInput interfaces. These pickling interfaces are shown in Figure 2.

```
interface ObjectOutputextends java.io.DataOutput
{
  // Write an object, array or String to the stream
  public void writeObject(Object o)
        throws IOException;
}

interface ObjectInputextends java.io.DataInput
{
  // Reads an object from the stream.
  public Object readObject()
        throws IOException;
}
```

Figure 2: Output and Input Interfaces

Pickling Process

To pickle an object, an ObjectOutputStream is created with an OutputStream to which the serialized form is written. Then, for primitives, the methods of the class DataOutputStream, such as writeInt or writeUTF, can be used. For objects, the writeObject method of the class ObjectOutputStream is used.

The writeObject method pickles the specified object and traverses its references to other objects in the graph recursively to create a complete serialized representation of the graph. The pickle of each object consists of the pickle of the class of the object followed by the object's fields.

Within a pickle stream the first reference to any object results in the object being serialized and the assignment of a handle. Subsequent references to that object record only the handle. Using object handles preserves sharing and circular references that occur naturally in object graphs. Repeat references to an object use only the handle allowing a very compact representation. Each field of the object is written appropriately depending on its type. Fields declared static or transient are not pickled.

The state of the object is then saved class by class from the base class through the most derived class. For each class the special methods are called, if they are defined by the class, or uses the default mechanism to pickle the fields of the object.

The client interfaces to write objects are implemented by the ObjectOutputStream class shown in Figure 3.

```
public class ObjectOutputStream
  extends java.io.DataOutputStream
  implements ObjectOutput
{
  // Creates a new context for an output stream.
  public ObjectOutputStream(OutputStream out)
        throws IOException {...}

  // Write an object, array or String to the stream
  public final void writeObject(Object o)
        throws IOException {...}

  // Called for each class written to the stream.
  protected void annotateClass(Class classs)
        throws IOException {...}

  // Called for each object written to the stream
  protected Object replaceObject(Object obj)
        throws IOException {...}
}
```

Figure 3: ObjectOutputStream Interface

Unpickling Process

To retrieve objects an instance of ObjectInputStream is primed with the stream containing the serialized representation. Primitives and objects are read from the stream in the same order as they were written. The readObject method operates recursively, retracing the sequence generated by writeObject, and following references in a corresponding manner. It creates a new instance for every object in the pickle and restores the object with inter-object references preserved.

When a new object is to be read from the stream, the ObjectInputStream reads the class of the object, creates a new instance of it and adds a mapping from the handle in the pickle to the new object. If the same handle is encountered later in the pickle, it returns a reference to the corresponding unpickled object.

The state of the object is then restored class by class from the base class through the most derived class. For each class the special methods are called, if they are defined by the class, or uses the default mechanism to unpickle the fields of the object.

The client interfaces to read objects are implemented by the ObjectInputStream class shown in Figure 4.

Pruning Object Graphs

There are two mechanisms for pruning the graph of objects to be pickled. In the first, the transient keyword is used to mark fields to indicate that they are not part of the persistent state of an object. The default pickling mechanism will not use transient fields.

The second form uses the specials mechanism, in which the programmer explicitly saves only that part of the object's state that is important. This usage is quite a bit more flexible and gives the programmer direct control over what is saved. The programmer may conditionally prune the graph by writing only some of the references to other objects. During unpickling the equivalent state can be recreated.

```
public class ObjectInputStream
  extends java.io.DataInputStream
  implements ObjectInput
{
  // Creates a new context for a input stream.
  public ObjectInputStream(InputStream in)
      throws IOException {...}

  // Reads an object from the stream.
  // The result should be typecast as desired
  public final Object readObject()
      throws IOException {...}

  // Locate the local class for the named class.
  protected Class resolveClass(String classname)
      throws IOException, ClassNotFoundException
      {...}

  // Called when an object has been unpickled.
  protected Object resolveObject(Object obj)
      throws IOException {...}
}
```

Figure 4: ObjectInputStream Interface

Class Specific Pickling

Special methods can be used to override the default pickling/unpickling mechanism on a per-class basis. These special methods are needed in cases where the class itself knows best how to save the persistent state or when the default mechanism cannot or does not save the needed state. For example:

- A class that defines a large data structure having a more compact representation would be a good candidate for writing special pickling methods.
- In the case of a native implementation with native state, the default pickling mechanism cannot access the state and appropriate specials would be needed to save the object's state.
- If the state of a class is a function of the local operating environment, then it should pickle itself with the information that will allow the unpickling to reestablish that state relative to the new local environment.

The specials interface is defined by three methods with the signatures shown in Figure 5.

```
// Write this classes fields to the stream.
private void writeObject(ObjectOutputStream out)
      throws IOException;

// Read this classes fields from the stream.
private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException;

// Called when unpickling is aborted.
private void readObjectCleanup(ObjectInputStream
                                            in)
      throws Exception;
```

Figure 5: Special Method Signatures

The special methods are private and can be called only by the pickling/unpickling run-time. If this were not so, it would be possible to call the writeObject or readObject methods to examine or modify the private state of objects. This would be a serious threat to the integrity of objects.

A class takes over pickling for itself by implementing both the writeObject and readObject methods. An exception is raised by the pickle stream if only one is implemented. Having only one of these methods would not make sense because the default mechanism cannot read data written by the special method.

The readObjectCleanup method is useful if a readObject method has taken some action that would need to be reversed if the unpickle of the graph fails. Usually when an unpickle fails, none of the objects are referenced (except by the pickle stream) and these will be garbage collected when the stream itself is reclaimed. However, if a readObject method has created references in the environment to an object in the incomplete graph, those references must be removed. A readObjectCleanup method is needed in this case to remove the references to the object so that it may be garbage collected.

Specials have the following call semantics:

- Specials are invoked at each level in the inheritance hierarchy starting from the base class and proceeding to the most derived class. If no special is defined for a class, the default mechanism is used.
- The special for a class is only responsible for pickling and unpickling its own fields. It cannot invoke specials in other classes and it cannot prevent other classes from being pickled.
- As with constructors, a readObject method can assume the current object has the type of its declared supertype. It should not, however, rely on the behavior of dispatched methods that might access the fields of its (yet to be unpickled) subclasses. This ordering allows the readObject method, if necessary, to override the actions of its superclasses with respect to public and protected fields.
- If a special raises an exception, the top-level operation is aborted with an exception. Typically this is done by classes that do not wish to be pickled.

Abort Processing during Pickling/Unpickling

The pickling and unpickling process involves many objects and interactions with the underlying system. Since many exceptional conditions may occur, some discipline is needed to handle them and define the state of the stream. The two usual cases are exceptions caused by I/O errors on the target/source stream or exceptions raised by pickle specials to prevent themselves from being pickled. These exceptions are caught by the pickle stream and mark the stream as aborted. Any subsequent attempt to write to the stream will fail by throwing an exception. Pickle special methods cannot override this behavior since the stream is marked aborted on receipt of the first exception and the stream will check this condition after every special terminates and rethrow the exception.

A pickle stream that has had an exception raised is not left in a usable state since an unknown amount of the object and the referenced graph has been pickled or unpickled. Due to the recursive and nested structure of the pickle no statement can be made about what data is in the pickle except that it is incomplete and unusable. It is up to the caller to determine how to continue.

Extending Pickling via the Subclass Interface

ObjectOutputStream and ObjectInputStream can be extended to allow per class and per object data to be pickled.

As every object is pickled, its class must be identified in the stream. This is done by pickling the class itself, using its name and fingerprint. When a class object is pickled, it may be desirable to pickle additional information for the class. For example, if a class was loaded from the network, saving the network address of the code in the stream will allow the class to be loaded on demand during unpickling. The annotateClass method is called after the class name and fingerprint have been pickled. Within this method additional information can be written to the stream.

During unpickling, the class name and fingerprint are retrieved from the stream and the resolveClass method is invoked. The resolveClass method needs to find the named class and return it. It should read any additional information written by the corresponding annotateClass method from the stream and load the class as needed. By default resolveClass just invokes Class.forName. The fingerprint, described in more detail later, is needed to confirm the semantic and structural equivalence of the class used to pickle the object as compared to the locally available class.

Another need that arises is to be able to substitute one object for another during pickling. To make this substitution possible ObjectOutputStream supports a protected replaceObject method. A subclass can define this method to substitute an alternate object. The replaceObject method is called once for each object; it may return either a substitute object or the original. Care must be taken to replace the object with a compatible object or later unpickling will be aborted with a ClassCastException when the incompatible object is assigned to a field or array element. Null may be returned by replaceObject but should be used carefully since setting references to null in arbitrary classes may cause unexpected exceptions in methods that act on the reconstructed graph.

Whenever possible, the writeObject and readObject methods should be used to pickle objects in an appropriate form. However, in cases where the objects implementation is not available, subclasses of the pickling implementations can be created to substitute objects to do a dynamic substitution. For example, objects that are instances of FontData could be replaced by the writeObject method with instances of a FontName class. The corresponding resolveObject method could replace instances of FontName with the corresponding local FontData class. Object substitution during pickling and unpickling can be a very powerful tool for dynamic binding of objects that are part of the environment.

Default Pickling of Objects

The pickling of objects is driven by the class of the object. Strings, Arrays, and Class objects are handled with specific mechanisms, described below, either because they have specific representations or are of special significance to pickling. For all other object types the default pickling mechanism is used.

The pickle stream implementations interpret the class information dynamically to locate and access the fields of an object. Private native methods access the class meta information and fields as follows:

- The fields of the object, except for static and transient fields, are put in canonical order (sorted) so as to be insensitive to reordering of declarations.
- The sorted fields are then written to or read from the stream dispatched by their signatures.
- Fields of primitive types invoke the corresponding DataOutput and DataInput methods.
- Arrays, Strings, and objects invoke the writeObject method on ObjectOutputStream, effectively recursing.
- During unpickling, new objects are created and assignments to each field are type checked.

Pickling and Unpickling of Class Objects

The integrity of the type system is maintained by treating objects of type Class specially and putting information in the pickle to correctly match the type of the object in the pickle with the class available during unpickling. This type matching is done using *secure fingerprinting*. Whenever a Class object is pickled, only its name and secure fingerprint are written to the stream. The details of the fingerprint are described in the section on FingerPrinting Java types.

Correct identification of classes is crucial since the structure of the stream is derived from the definition of the class. Any change of names or types of fields in the class would cause the stream to be misinterpreted. Making the streams self describing would greatly increase the overhead.

ObjectInputStream verifies that the structure implicit in the stream is the same as that defined by the currently available class. This allows primitive fields to be read without additional type checks. For fields that contain references, every assignment of an unpickled object is type checked against the type of the field and for each assignment to an array element.

Pickling of Arrays and Strings

Strings and arrays are treated specially in pickles in order to deal with their specific representations. Both are objects so reference sharing must also be preserved.

Pickling of array objects writes the array signature, length and then iterates over the contents of the array to pickle each element according to its type.

Reading arrays is complementary to writing. First the array signature and length are read, and the first level index is created. Each of the elements is then read and assigned. If the elements are arrays the process recurses naturally. For arrays of primitive types, the functions of DataInputStream are used to read the elements. For object types, Strings, sub-arrays and class objects, readObject is used. This design maintains any potential reference sharing to arrays and subarrays.

Strings are pickled in their Universal Transfer Format (UTF) form but require special handling since they have a native implementation.

Modal Pickling

In practice there are many different situations in which pickling is required, and a different scheme may be required in each. Although there is just a single set of specials for each class, modal pickling can be supported by having the specials switch on the type of the stream which invokes them. Whenever modal-pickling is necessary, appropriate subclasses of the pickling streams would be defined. This allows for forward compatibility, as in any subtyping scheme.

An example of modal pickling is its use in a remote method invocation mechanism[8]. Remote objects are defined by Java interfaces. References to remote objects are declared using these remote object interfaces, not the implementations that support the interfaces. In remote method invocation two types of objects need special handling, the surrogate for a remote object and the remote object itself. Pickling of the surrogate is handled using a writeObject special to preserve the remote reference information. However, a reference to the remote object implementation should not be pickled as itself since that would include all of its implementation state. Instead, it should be pickled as a surrogate object with the matching interfaces and the necessary remote reference information. The marshaling subclass of the pickle output stream is responsible for detecting the implementation object and finding or creating the appropriate surrogate. Both the surrogate and the remote object implementation support the remote object interfaces, and as such both are type compatible with the field and array elements of the defined remote object interfaces.

4 Fingerprinting Java Types

A fingerprint is a concise representation of a piece of data, typically created by a hashing scheme that satisfies the following:

- Low collision: Two pieces of data will hash to the same pattern with extremely low probability (low enough to be ignored).
- Rehashable: Hashes themselves can be input to the hash algorithm.
- Fixed length: The representation is of fixed length.

Java types are fingerprinted by computing a shallow fingerprint for the class and each of its superclasses up to java.lang.Object. The shallow fingerprint consists of:

- The class name encoded in Universal Transfer Format (UTF).
- The class access flags as an integer, (such as public, interface,...).
- The sorted list of interfaces supported by the object encoded as UTF.
- The sorted list of fields, including field name, signature and access modes (such as public, static, protected, private, transient...).
- The sorted list of methods, including method name, signature and access modes (such as public, private, static,...).

The shallow fingerprints are computed using the NIST Secure Hash Algorithm (SHA)[7]. The fingerprint is the rehash of the concatenation of the hashes for each class up the supertype chain. The result is the secure hash of the class. The hash will differ if any changes are made to any of the classes.

This "shallow" fingerprinting scheme provides more assurance than merely hashing the name of the type. Types need to be equivalent in structure as well as name. However, the cost associated with a scheme that hashes the entire hierarchy is not justified, given that the rest of the Java system environment uses name equivalence, and that all classes are verified for consistency as they are loaded by the class loader.

The shallow fingerprinting scheme provides the *same* assurances as a deep fingerprinting scheme if the fingerprints of all referenced object types are matched as well (transitively). Classes are type matched only when an object of that type is pickled/unpickled. The type matching explicitly includes the signatures of all of the supertypes. This insures the structural equivalence of the object and corresponding sequence of bytes in the stream needed to unpickle the object. Other classes this object depends upon will be type matched when the first object of that class appears in the pickle. Objects are polymorphic so the actual types must be represented in the pickle as well as the declared type. The declared types are included in the fingerprint as one of the supertypes of the actual type.

Pickling the `null` reference is interesting because it carries no type information. However this does not present a problem for pickling. The usual interclass dependency checking performed by the class verifier will validate the classes effectively so that they may operate correctly.

Secure Hash Algorithm (SHA)

The National Institute of Standards and Technology (NIST) designed the Secure Hash Algorithm (SHA) for use in the Secure Hash Standard (SHS), to support the Digital Signature Algorithm (DSA). SHA produces secure 160 bit hash values from data of arbitrary length. SHA provides all the guarantees mentioned above. SHA is described in detail in [7]. We use SHA Version 3.

Java Type Specifiers

The strings and values that are input to the hash algorithm are all defined by the formats and constants defined by the Java Virtual Machine Specification[5]. Class names, method names, method signatures, field names and signatures are strings. Method and field signatures use the same encoding as defined by the type system in the Java Virtual Machine. The access flags for the classes, methods, and fields use only the significant bits as defined in the Java Virtual Machine specification.

5 Structure of Pickles

A *pickle* is a representation of the state of a sequence of values (objects, arrays, Strings, Classes, or primitives) corresponding to a sequence of writes. It begins with a magic number of 16 bits and a version number of 16 bits representing the version of the pickling run-time which produced the pickle. The unpickling run-time should ensure that it can process the pickle before it proceeds.

Primitive types are written to the pickle without any preamble or type-checking information by the methods of java.io.DataOutputStream, a core class that encodes primitive types into a stream in a portable fashion. Those methods are also used to write out the fields and values of objects. Reads correspondingly make direct use of the methods in the standard java.io.DataInputStream class. Since such types have a fixed format which is common to all implementations of the Java System, there is no reason to add type checking information, which helps keep pickles small. Since these are not reference types, the caller is required to know the type of the value before reading it for the purposes of assignment. Consequently a preamble containing type information is not needed.

Handles are assigned to objects written to the stream. They are assigned in ascending order using 32 bits and have a base offset value to make them more unique in the stream. However, these handles do not appear in the stream except when a reference is made to a previously pickled object. The writeObject and readObject methods are assumed to be synchronous in assigning handles. This saves space in the pickles, especially in the cases where there are no back references (as occurs in argument marshaling).

Markers are inserted into the stream to validate the alignment of the stream. Markers are 16 bit counters that can wrap around as necessary. Markers are put into the stream at the end of each object, Array, and Class. Markers are also inserted following each call to a special writeObject method. This improves the probability that errors in class specific writeObject and readObject combinations will be detected as soon as possible.

The following description of the pickle format is not rigorous but will provide a flavor for the layout of objects in the pickle. Objects, arrays, strings, and classes that are being written to the pickle for the first time are formatted as follows:

```
<OBJECT object>:= <CODE for OBJECT>
  <Reference to Class of o>
  <Implicit assignment of handle for o>
  <State of the object's base class>
  [Marker if state was not saved by default]
  ...
  <State of the object's most derived class>
  [Marker]

<String object>:= <CODE for String>
  <Implicit assignment of handle for object>
  <UTF of String>

<Array object>:= <CODE for ARRAY>
  <UTF of classname of array>
  <Implicit assignment of handle for object>
  <Length of first index of array (32 bits)>
  <length> objects
  [Marker]

<Class object>:= <CODE for CLASS>
  <Classname in UTF format>
  <20 byte signature of class>
  <Implicit assignment of handle for object>
  <Call to protected annotateClass method>
  [Marker]
```

Other references to objects are encoded as either NULL or as reference to a previously pickled object. The encoding is:

```
<Object null>:= <CODE for NULL>

<Object object>:= <CODE for REFERENCE>
  <Handle as previously assigned to the object>
```

6 Integrity Validations

Since the source of code commonly present in the Java runtime environment is unknown and it may initiate or be involved in pickling, the pickling mechanism needs to be made as tamper proof as possible.

In the attempt to make the pickling mechanism robust the Java language's inherent safety features have helped maintain the integrity of classes.

- The pickling classes provide read and write methods which are declared final and hence can be trusted. This allows the programmer of a class to be sure that the fact that ObjectOutputStream and ObjectInputStream can be subclassed does not mean the specials will behave differently in different circumstances.
- Similarly the programmer of a class can be sure that subclasses of the class will not be able to prevent their specials from being called when an instance is being pickled/unpickled. This allows the special to raise an exception if it wishes, to enforce the class's right not to be pickled.
- The pickling run-time cannot be replaced with another implementation. This is prevented by the class loader and security manager. For the same reason, the native code which provides internal access to objects is only accessible privately to ObjectOutputStream and ObjectInputStream classes and cannot be accessed otherwise.
- It is not possible to corrupt objects using the readObject method, i.e. it is not possible to get the unpickling run-time to reinitialize a pre-existing object from a pickle-stream. This is because the special methods are private and ObjectInputStream only calls special methods on objects it has created.

Remaining Concerns about Data Integrity

Not all concerns have been eliminated however:

- Pickles have a publicly known format and can be tampered with. Just as sensitive fields of the object must be marked private, those same fields must not be pickled carelessly. Marking them transient is an easy mechanism to keep them out of the pickle. If sensitive data must be pickled there is no solution short of encryption with a shared key that will protect the information.

- Fingerprinting compares classes for both structural and name equivalence. This does not guarantee that the code implementing the class does what it is expected to. Support for identifying and authenticating classes is a basic requirement and is outside the scope of pickling.
- It is possible to create an object that could never exist, by unpickling it from a concocted pickle. Since all classes in the hierarchy need to initialize themselves, specials need to be satisfied with the data they read, and marker boundaries need to be respected, this is not an easy thing to do. However, it is technically possible and can produce an object that violates internal invariants between data fields which may be integral for the object's correct operation. This potential for error can be mitigated to some extent by having read-specials explicitly test invariants.
- It is possible to violate opaqueness by using unpickling to "look" inside objects to examine private state. This is typically a concern faced only by private, critical classes. The class can prevent access by marking sensitive fields transient, by using the specials mechanism to control access or raising exceptions as appropriate.

Guidelines for Safe Pickling of Sensitive Classes

When writing a class that provides controlled access to resources, care must be taken to protect the mechanisms that access those functions. During unpickling (by default) the private state of the object is restored. For example, a file descriptor is a handle that provides access to an operating system stream. Being able to forge a file descriptor would allow some forms of illegal access, since restoring state is done from an insecure stream. To avoid compromising access control, the related state of an object must not be restored from the pickle or it must be reverified by the class. Use one of the following techniques to protect sensitive data in classes:

- The easiest technique is to mark fields that contain sensitive data as "private transient". Transient and static fields are not pickled or unpickled. Simply marking the field will prevent the state from appearing in the pickle and from being restored during unpickling. Since pickling and unpickling (of private fields) cannot be taken over outside of the class the classes transient fields are safe.
- Particularly sensitive classes should not be pickled at all. To accomplish this writeObject and readObject methods (with signatures as described above) should be implemented to throw a NoAccessException passing its class name. Throwing an exception will abort the entire pickling process before any state from the class will be pickled or unpickled.
- Some classes may find it beneficial to allow pickling/unpickling but specifically handle and revalidate the state as it is unpickled. The class should implement writeObject and readObject methods to save and restore only the appropriate state. If access should be denied throwing a new NoAccessException will prevent further access.

7 Future Work

While the current implementation is robust and extensible some additional work is needed. We have not looked at the performance aspects of this scheme but believe that it needs to be characterized and improvements made. Also, the current recursive traversal is suitable for only modest size graphs and will need to be extended to accommodate very large graphs.

No attempt has been made in this work to address the evolution of classes. The general object versioning problem has taken many forms and is very complex. We are investigating what is needed in pickling to support the evolution of classes.

The security of private information is of vital concern to the Internet community. The current implementation requires the programmer to go beyond just putting data in private fields to keep data secret. More conservative approaches are needed for production systems.

Availability

Java Object Serialization will be released with JDK 1.1. Early access versions of this system can be obtained from http://java.sun.com.

References

[1] Arnold, Ken, and James Gosling, *The Java Programming Language*, Addison-Wesley (1996).

[2] Birrell, Andrew, Michael B. Jones, and Edward P. Wobber, *A simple and efficient implementation for small databases*, Digital Equipment Corporation Systems Research Center Technical Report 24 (1987).

[3] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, *Network Objects*. Digital Equipment Corporation Systems Research Center Technical Report 115 (1994).

[4] Gosling, James, and Bill Joy, Guy Steele, *The Java™ Language Specification*, in preparation.

[5] Herlihy, M and B. Liskov, *A Value Transmission Method for Abstract Data Types*, ACM Transactions on Programming Languages and Systems, Volume 4, Number 4, (1982).

[6] Lindholm, Tim and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley (1996), in preparation.

[7] Schneier, Bruce, *Applied Cryptography*, John Wiley & Sons, Inc (1994).

[8] Wollrath, Ann and Roger Riggs, Jim Waldo, *A Distributed Object Model for the Java™* system, Proceedings of the USENIX 2nd Conference on Object-Oriented Technologies and Systems (1996).

Authors

Roger Riggs (roger.riggs@Sun.com) is a Staff Engineer at JavaSoft, working on large scale distribution and reliable distributed systems. Prior to joining the group, he worked on client-server text retrieval and CORBA based information retrieval, and scalability issues on the Web.

Jim Waldo (jim.waldo@Sun.com) is a Senior Staff Engineer at JavaSoft, working in the area of reliable large-scale distributed systems. Prior to joining Sun, he worked in distributed systems and object-oriented software development at Apollo Computer (later Hewlett Packard), where he was one of the original designers of what has become the Common Object Request Broker Architecture (CORBA).

Ann Wollrath (ann.wollrath@Sun.com) is a Staff Engineer at JavaSoft, working in the area of reliable large-scale distributed systems. Prior to joining Sun, she worked in the Parallel Computing Group at the MITRE Corporation investigating optimistic execution schemes for parallelizing sequential object oriented programs.

Krishna Bharat (kb@cc.gatech.edu) is a Ph.D. student at the College of Computing, Georgia Tech, working in the area of environment and system support for constructing distributed user interface applications. This work was begun when he was a research intern at Sun Microsystems Laboratories (East). He plans to finish his dissertation in June '96, and will be joining the research staff of Digital, Systems Research Center.

Highly Concurrent Distributed Knowledge Objects

K.L. Clark, T. I. Wang
Department of Computing
Imperial College
London
{klc,tiw}@doc.ic.ac.uk

Abstract

This paper introduces a distributed object oriented logic programming language, DK_Parlog^{++}, in which each object, in addition to the normal procedural methods for accessing and updating state components, also has knowledge methods in the form of Prolog rules. The language is an OO extension of a distributed, multi-threaded, logic programming system. Encapsulation is by class definitions linked via single inheritance. Procedural method invocation is via asynchronous message send to the unique object identifier. Knowledge method invocation is via synchronous remote procedure call. Classes are also objects - they have their own state components and methods. An application in D_Parlog^{++} consists of a collection of concurrently executing objects(classes and instances) distributed over a local area network.

The knowledge methods can be public or private, and can be dynamically modified by the procedural methods of the object using Prolog style knowledge base manipulation primitives. The dynamic knowledge of an object can be used as a declarative representation of part of its state.

This paper assumes some familiarity with concepts of concurrent object oriented programming and logic programming, ideally concurrent logic programming, but familiarity with Prolog will probably suffice.

1 Introduction

Shapiro and Takeuchi[ST83] first demonstrated the potential of combining Concurrent Logic Programming (CLP) and OOP to form a new programming paradigm. They investigated the possibility of implementing active objects, with encapsulated state, as processes in a CLP language. In their model, an object is implemented as a tail-recursive process that passes the updated state components of the object as arguments in the recursive call.

Following their model, several language proposals such as Polka[Dav89], Vulcan [KTMB87] and A'UM[YC88] have since emerged. Each is essentially syntactic sugar for defining the CLP process that will implement each instance object.

Most of these languages use a class declaration (or an equivalent) just as the vehicle for defining the methods of the instances of the class. The class declaration is compiled into a procedure definition in the CLP language. Creating an instance of a class is then just syntactic sugar for a direct invocation of this procedure as a recursive process. In this approach, no global information about these created instances is automatically kept. Thus, no managing operations on these instances can be conveniently performed. For example, one cannot easily broadcast a message to all the members of the class. In addition, objects cannot be given public names, so in order for one object O to send a message to another object O', O must be passed the identity of O' in a message, or be given the identity as the value of one of its state components when it is created. Finally, inheritance is generally implemented as message delegation, and for each created instance of a subclass there will be a string of processes forked, one for each level in the class hierarchy. This is perhaps acceptable in a non-distributed system, but is not accepted if the different processes might reside on different machines. We have adopted a somewhat different approach in constructing our OO extension of a distributed version of the concurrent logic programming (CLP) language Parlog [CG86].

In our approach, classes are active objects, they have there own methods and, like the class instances, are implemented as Parlog processes. Classes are also the main mechanism for distributing the computation. We just load different class objects onto different machines of a local area network. All the active instances of a class will, by default, reside on the same machine as the class, but an inheriting class, and hence all its instances, can reside on another machine. Inheritance of class methods is implemented using delegation, but inheritance of instance methods is implemented using code copying at compile time as in Smalltalk[GR83]. So instance level method inheritance, even when the superclass resides on a different machine, does not require a delegation communication between the machines.

All object methods, whether for class or instance objects, are invoked by sending a message to the object denoted by its unique object identity. A class object's identity is its public name, such as **bank**, given in the program. An instance identity can either be programmer assigned, or system generated. The fact that instance objects can have public names is another distinguishing feature of the language. **M => O** sends message M to O asynchronously.

The language is highly concurrent. All objects can process their stream of incoming messages concurrently. That is, as soon as a message is accepted, and the method for the message is activated, the object is able to accept the next message. In addition, each of the actions of a method can be executed concurrently. So DK_Parlog^{++} objects can exhibit the high degree of internal concurrency of actors [AH87]. On the other hand, the actions of a method can be sequentialised if need be, and much more easily than in the actor paradigm. It is simply a matter of defining the method as a sequential, rather than parallel, conjunction of actions.

Adding knowledge to objects A major shortcoming of the CLP based OO languages is that method calls can only return one solution. This is because of the committed choice non-determinism of the underlying CLP language.

However, to many, logic programming is identified as the definition of relations, and with queries that return multiple solutions. So, as well as having the normal single solution *procedural* methods, every object in D_Parlog^{++} can also have *knowledge* methods, expressed as Prolog clauses. These knowledge methods can be further subdivided into public and private knowledge methods. In contrast to the procedural methods, which cannot be changed, knowledge methods can be dynamically modified during the lifetime of an object.

The public knowledge component of an object O can be queried from any other object with a query the form **O?Q**. This is a synchronous remote procedure call.

Since a new query on an object's public knowledge can arrive before it has answered the last query, an object will accept and concurrently execute a new query as soon as the preceding query evaluation has commenced. For each object there is a queue of queries to its public knowledge, as well as the queue of messages invoking its procedural methods.

The caller receives an answer as soon as the first solution to its query is found. If the query was from in a procedural method of the caller, this is the end of the query computation. However, if it was from a knowledge method of the call, backtracking within this caller knowledge method may require further solutions of the query. A distributed backtracking scheme handles the request and delivery of successive solutions between caller and callee object.

Finally, the dynamic knowledge of an object gives us an alternative way of encoding state information about the object. The special knowledge manipulation primitives, which can only be called from a procedural method of an object or its class, allows the runtime manipulation of the dynamic knowledge of an object. A knowledge update call enters the query queue for an object, however, to maintain consistency of evaluation, the knowledge update will be only be executed when all previously accepted queries have completely terminated. No new query will be accepted until the update has terminated. Evaluation of the knowledge queries is handled by IC-Prolog II[CC93], a multiple-threaded Prolog system[CC93] which is linked with the underlying Parlog system of DK_Parlog^{++}. Full details of the implementation of DK_Parlog^{++} are given in [Wan95].

2 An introductory example

Our first example comprises the two class definitions given in Program 2.1. In this example there are no program defined class variables or methods (although as we shall see there are system added class variables and methods) and there are no knowledge methods.

The Superclass - account The first class, called **account**, defines the basic information and some simple operations for a bank account.

The instance state component includes variables for the name of the account holder, the balance of the account and the accumulated interest over a certain period. Two methods are shown in the instance definition. The first is a simple inquiry method which returns the present balance of the account, and the second is a method to calculate, on request, the interest earned by the account. Notice that the second instance method *calculate_interest/0* uses a state component `InterestRate` which is not defined yet. The assumption is that this will be an instance variable of each subclass of **account** and that only instances of these subclasses will be created. In this example, there is no particular advantage to not having the state component `InterestRate` in **account**. It is done just to demonstrate the flexibility of using inheritance for organizing classes. However, using the same idea for methods is quite useful. We can invoke an as yet undefined method using a **self** communication and different subclasses can implement the method in different ways.

```
class account at laotzu with
{
 instance_definition
    states
      Holder,Balance:=0,AccInterest:=0.
    methods
      balance(BAL) -> BAL = Balance.

      calculate_interest ->
         AccInterest := AccInterest +
                InterestRate * Balance.
}.
class current isa account at lipo with
{
 instance_definition
    states
      InterestRate, CreditLimit.
    methods
      credit(AMOUNT) ->
            Balance := Balance + AMOUNT.

      debit(AMOUNT, ANSWER):
          Balance >= AMOUNT ->
            ANSWER = AMOUNT,
            Balance := Balance - AMOUNT.

      debit(AMOUNT, ANSWER):
          VitualBalance is Balance +
                           CreditLimit &
          VitualBalanc >= AMOUNT  ->
              ANSWER = AMOUNT,
              Balance := Balance-AMOUNT.

      debit(AMOUNT, ANSWER) ->
            ANSWER is Balance+CreditLimit,
            Balance := - CreditLimit.

      collect_interest ->
           Balance := Balance+AccInterest,
           AccInterest := 0.
}.
```

Program 2.1 Current Account class through inheritance

The Subclass - current Next, the program defines another class **current**, which inherits the **account** class.

Every class does have some systen added class variables and methods. For example, in each class there is a class variable `Members` which holds the identities of all the current instances of the class. It also has a visible method for accessing the value of this class variable and invisible methods, invoked on instance creation and termination, for updating it.

In **current** class, two instance state variables are defined, one is the expected `InterestRate`, and the other is the `CreditLimit`, which will hold the maximum amount an account can be overdrawn. Thus, for creating an instance of the **current** class, one might use a create message as:

```
create(AccountID,0.02,400,
      'Debbie',2000,_) => current
```

Notice that initial values for the bottom object are given first, but also that all state components, including the inherited ones, need to be mentioned in the create. The value of the last state variable, `AccInterest`, is given as '_'. This indicates that the default value given in the class definition should be used. The 2000 given for the initial value of `Balance` overrides its default value.

In addition to the two methods inherited from its superclass, instances of the **current** class have five more methods. The `credit/1` method is very simple, and just adds the `AMOUNT` into the balance of the account. The next three are all mehtods for handling a `debit/2` message but each deals with it differently depending upon the current value of the state variable `Balance`. The first two have the structure

```
message_pattern : test -> actions
```

with the : introducing the test to be applied before that method is used. Notice that the last method for `debit/2` does not have a test. It is the default rule that will be used only if the previous two rules are not applicable. The first method treats the case that the current `Balance` is sufficient for the withdrawal. The withdrawal is granted by assigning the value of `AMOUNT` to the `ANSWER` variable and the balance is reduced accordingly. If the balance is insufficient, but the shortage can be covered by the credit limit, the withdrawal is also granted with the balance going into a debt status. The sum of the balance and the credit limit is, however, the maximum amount for a withdrawal. If the credit limit cannot cover the shortfall then the amount withdrawn is reduced to `Balance + CreditLimit` (the default method). Finally, there is a method `collect_interest/0` for transferring the accumulated interest into the balance and clearing the accumulated interest. As can be seen in these methods, instance state variables declared in the superclass can be accessed directly in the instance methods of the subclass.

Finally, note that the **account** class is specified as residing on the machine laotzu, while the **current** class is on the machine lipo.

3 Knowledge Representation in Objects

Static Knowledge is given as **Knowledge Methods** alongside the procedural methods. Dynamic knowledge is asserted into objects either when the object is created or after it has been created.

Knowledge declared in the class definition section of a class is local to the class process, while knowledge declared in the instance definition section of a class is global to all instances, i.e. it is automatically possessed by any instance when it is being created. Knowledge asserted into an object is local to the object itself. All instance level knowledge methods declared in the class are inherited when an inheritance link is declared. As in L&O[McC92], knowledge inheritance can be inclusive or overriding.

3.1 Static Knowledge Declaration

A knowledge method is just a Prolog clause for a predicate **p/k** of the form:

CallPattern [:- *Actions*].

The *CallPattern* is an atomic formula of the form:

$\mathrm{p}(t_1,\ldots,t_k)$

Knowledge methods are distinguished from procedural methods by use of the Prolog *if* connective :- instead of the connective ->. *Actions* is a conjunction of Prolog conditions, and constitutes the body of the clause.

To query the knowledge in another object **O** we use the call **O ? Q**. A simple example is:

customer?the_best(Banks,Prof,Type,BestBank,
        BestRate,InitCredit).

It is executed as a remote procedure call. Invoking a knowledge method not defined in an object simply fails the invocation.

As with procedural methods, the next call to a knowledge method can be accepted and executed by an object whilst the previous call is still being processed. There are exceptions to this immediate acceptance of the next call. Knowledge update calls will be delayed until all existing query calls have terminated. This will be discussed more fully later.

Private knowledge can be given as Prolog clauses in a special code section of the object. This section can also include Parlog definitions, which can be used as private procedural methods.

```
class customer with
{
 class_definition
  methods
   ..... %Other methods ...............
   the_best(Banks,Prof,Type,BestBank,
                    BestRate,InitCredit):-
      findall(
          (Bank,Rate,Init),
          (on(Bank,Banks),
           Bank?rate(Type,Prof,Rate,Init)),
          ANS),
       find_the_best(ANS, BestBank,
                     BestRate, InitCredit).
 instance_definition
  states  Name.
  methods
   which_bank(Prof,Type,Bank,Rate,
                          InitCredit) :-
     members(Banks) => banks,
     class?the_best(Banks,Prof,
       Type,BestBank,BestRate,InitCredit).
     :
     :  %Other methods
 code_definition
   find_the_best(Tuples,BestBank,
                       BestRate,InitCredit):-
    ..%definitions omitted for simplicity...
}.
```

Program 3.1 Knowledge methods in a customer class

A simplified class structure which declares a class **customer** for a financial advice program is shown in Program 3.1 to demonstrate the use of knowledge methods. The *which_highest/6* knowledge method in the class definition is meant to find out which bank offers the best interest rate on a certain type of account for a certain type of customer. It receives from its first three arguments a list of bank names, the profession of a customer and the type of the account the customer intends to open, then uses the

Prolog style *findall/3* predicate to find out, of all the banks, the interest rate for the specified types of account and profession and the initial credit required for the type of account, and finally passes the information to a private procedure to figure out which bank should be recommended. The use of *findall/3* predicate fully reflects the desire of the searching capability in an object. A knowledge access communication is used in the call of the *findall/3* predicate to access the interest rate and the required initial credit of an account type for each kind of profession, the semantics of such knowledge access operation will become clearer later.

The other instance level knowledge method *which_bank/5* uses a mutual communication to get the list of banks from the **banks** class.(*member/1* is a system added class method for accessing the list of current member of a class.) It then uses a *class* communication to access the knowledge of the best bank choice. Class communications can be used deliberately to access knowledge kept in the class level.

The purpose of Program 3.1 is to show how knowledge methods can be declared in both class and instance definition sections. Here, the functionality of the only *the_best/6* class knowledge method could have been implemented by a procedural method which invoked a private Parlog recursive procedure. However, morte generally, there might be several different ways of 'rules of thumb' for determining the best bank for for a type of customer and account in which case it is better to use Prolog style rules for encoding such advisory rules.

3.2 Dynamic Knowledge Assertion

While static knowledge represents long term information provided by an object, dynamic knowledge is object specific or updatable information. There are two ways for associating dynamic knowledge to an instance. One is to attach a sequence of knowledge methods to the create message sent to a class. When the instance is created, the sequence of methods will become the initial dynamic knowledge. The other is to use a special system predicate to assert a single knowledge method into an existing instance. To add dynamic knowledge to a class process, obviously only the second way can be use. There is no difference in accessing the two kinds of knowledge.

To signal what an object can have in its dynamic knowledge, a **dynamic declaration** must be included in either the class_definition or the instance_definition section of a class structure. The declaration is a list of *name/arity* terms giving the dynamic predicates. As an example, class **banks** declared in Program 3.2 is for spawning bank instances for the financial advising program in the previous section. It is simplified and its instance_definition section contains a **dynamic** declaration which specifies that *profit_margin/2, rate/4* are dynamic. There is no dynamic declaration in the class_definition section, which means no dynamic knowledge methods can be acquired by the class process. Following the class definition comes a statement for creating an instance of the **banks** class. The general format is:

create(*object_id* [, *arugments*])
with [{ *Knowledge Methods* }]
=> *class_identity* [**at** *machine_identifier*]

The optional **with** gives extra Knowledge Methods which are the dynamic knowledge methods local to the new object.

What is local to this created bank instance, which has the public name `barclays`, are three facts about the profit margins, set for different professions, above the base interest rate, and another three rules for providing the information on interest rates and minimum credits on different types of accounts. This information is given as clauses for the dynamic predicates `profit_margin` and `rate`. The instance with the name `midland`, created by the message send of 3.3, has different definitions for these dynamic predicates. Each bank has its own data and rules, its own *enterprise rules*.

A message send operation:

create(kate,'kate smith') => customer

will create an publically named customer instance from the **customer** class. Notice that `kate` is the public name of the object, whereas `'Kate Smith'` is the string value of the `Name` state variable held inside the object. A method invocation such as:

which_bank(student, current, Bank, Rate, InitCredit)
=> kate

will result in the following answer bindings:

Bank = midland
Rate = 3.1
InitCredit = 100

```
class banks with.
{
 class_definition
   states  BaseRate.
   methods
     new_base_rate(NBR) -> BaseRate := NBR.
     base_rate(BR) -> BR = BaseRate.
     :
     :  %Other methods
 instance_definition
   dynamic  profit_margin/2, rate/4.
    states   BankName, Manager.
    methods
       :
       :  %Other methods
}.
create(barclays, 'Barclays Bank Plc',jane)
with
{
  profit_margin(student, 0.4).
  profit_margin(general, 0.5).
  profit_margin(business,0.6).
  rate(current, Profession,Rate,InitCredit)
   :-base_rate(BR) => class,
     profit_margin(Profession, PM),
     Rate is BR - PM,
     InitCredit = 100.
  rate(select, Profession,Rate,InitCredit)
   :-base_rate(BR) => class,
     profit_margin(Profession, PM),
     Rate is BR - PM + 2.0,
     InitCredit = 1000.
  rate(tessa,Profession,Rate,InitCredit)
   :-base_rate(BR) => class,
     profit_margin(Profession, PM),
     Rate is BR - PM + 1.5,
     InitCredit = 20.
} => banks.
```

Program 3.2 A banks class definition and an instance of it

```
create(midland, 'Midland Bank Plc',peter)
with
{
 profit_margin(student, 0.3).
 profit_margin(general, 0.4).
 profit_margin(business,0.6).
 rate(current,Profession,Rate,InitCredit):-
     base_rate(BR) => class,
     profit_margin(Profession, PM),
     Rate is BR - PM,
     InitCredit = 100.
 rate(tessa,Profession,Rate,InitCredit):-
     base_rate(BR) => class,
     profit_margin(Profession, PM),
     Rate is BR - PM + 1.7 ,
     InitCredit = 25.
} => banks.
```

Program 3.3 Another instance of the banks class

One very important common characteristic of both kinds of knowledge is that state components can not be accessed directly from within a knowledge method. This may sounds like an inconvenience, however, can be compensated by two programming techniques. The first technique is to issue a self communication in a knowledge method to access the desired state components. Of course, an extra procedural method is needed for providing those state components. The second technique is to store the involved state information as dynamic knowledge rather than as the value of state variables, and to access them directly in the knowledge methods, we will discuss this concept more completely later.

4 Knowledge Inheritance

The inheritance policy for the knowledge methods is slightly different from that for procedural methods. In addition to the inclusive and overriding inheritance, there is also a differential inheritance.

4.1 Ordinary Inheritance of Knowledge - Inclusive

In general, if not specifically indicated, a class will inherit inclusively from its superclass all the instance and class level knowledge methods, which means that knowledge methods having the same *Predicate* as that of some methods in the inheriting class will not be overridden. Instead, they are aggregated into a same knowledge predicate and backtracking on such a knowledge predicate will extend to them. This procedure can be regarded as knowledge accumulation and ordinary declaration is used for this type of inheritance. For example, Program 4.1 declares a *wholikessbarclays* class which inherits the *customer* class in Program 3.1. It uses ordinary, thus inclusive, inheritance, therefore, if a knowledge access call which invokes the predicate *which_bank/5* of an **wholikesbarclays** instance fails to find the trusted bank provides an interest rate higher than 4.0, the execution will backtrack to the inherited method in class **customer** to find the best recommendation. Conceptually, in an **wholikesbarclays** instance, the predicate *which_bank/5* looks like:

```
which_bank(Prof,Type,Bank,Rate,InitCredit)
  :- self ? the_best([barclays],
       Prof,Type,Bank,BestRate,InitCredit),
  BestRate > 4.0.
which_bank(Prof,Type,Bank,Rate,InitCredit)
  :- members(Banks) => banks,
     self ? the_best(Banks,Prof,
       Type,BestBank,BestRate,InitCredit).
```

Notice the order of the clauses in the predicate. In the first clause, only bank 'barclays' is passed to the banks list for evaluating the best rate, and it is expected that 'barclays' should provide an acceptable rate higher than 4.0, otherwise, the second clause, which is actually inherited, will be used to evaluate the best rate.

```
class wholikesbarclays isa customer  with
{
 class_definition
   methods
     ..... %Other methods ...............
 instance_definition
  methods
  which_bank(Prof,Type,Bank,Rate,InitCredit)
     :- self?the_best([barclays],Prof,
           Type,Bank,BestRate,InitCredit),
        BestRate > 4.0.
       :
       :  %Other methods
       :
}.
```

Program 4.1 Inclusive inheritance of knowledge methods

4.2 Overriding Inheritance of Knowledge

Overriding inheritance of a class is declared with a different inheritance operator, **isa***, which indicates that knowledge methods in the superclass will be overridden if they have a same *Predicate* as that of some methods in the inheriting class. Backtracking on a knowledge predicate in a class will not extend to overridden methods, however, if required, super communications can still access these overridden methods. For example, Program 4.2 includes a **whotrustsbarclays** class which uses overriding inheritance when inheriting from **customer** class. An instance of it, upon receiving a *which_bank/5* knowledge access message, will just consult bank 'barclays' for the interest rate of a specific type of account.

However, if 'barclays' does not provide that type of account, the invocation will simply fail, and there is no chance that the execution will backtrack to the method declared in the superclass. Another *other_bank/6* method is also shown in the program to demonstrate how to use a super communication to call an overridden knowledge method.

```
class whotrustsbarclays isa* customer with
{
 class_definition
   methods
     ..... %Other methods ...............
 instance_definition
   methods
     which_bank(Prof,Type,
                  Bank,Rate,InitCredit) :-
       Bank = barclays,
       Bank?rate(Type,Prof,Rate,InitCredit).
     other_bank(Prof,Type,
                  Bank,Rate,InitCredit) :-
       super?which_bank(Prof,Type,
                  Bank,Rate,InitCredit).
     :
     :  %Other methods
}.
```

Program 4.2 Overriding inheritance of knowledge methods

4.3 Differential Inheritance of Knowledge

Differential inheritance of a class is declared with the ordinary inheritance operator but the name of the superclass is followed by a subtracting operator and a list of name/arity terms. Those superclass knowledge methods whose predicates are among the terms in the list are not inherited. This mechanism is the combination of the previous two inheriting styles, and the backtracking will extend to the inherited methods, but not to the subtracted ones. For example, in Program 4.3, a **loyaltobarclays** class is defined, in which the *which_bank/5* knowledge method in the **customer** class is excluded from the inclusive inheritance. Instead, it has a new *my_bank/5* method. Now, it will not even answer a *which_bank/5* message, although the *which_bank/5* knowledge method in the superclass is still reachable by a super communication from any of its methods. If an invocation to *my_bank/5* method fails to reach a conclusion because, say again, the 'barclays' does not provide certain types of account, the execution will simply fails.

```
class loyaltobarclays
   isa customer - [which_bank/5] with
{
 class_definition
   methods
     ..... %Other methods ................
 instance_definition
  methods
    my_bank(Prof,Type,
  Bank,Rate,InitCredit) :-
      Bank = barclays,
      Bank?rate(Type,Prof,Rate,InitCredit).
    :
    :  %Other methods
}.
```

Program 4.3 Differential inheritance of knowledge methods

Notice that all these different mechanisms do not affect the inheritance of procedural methods and state components, they apply to knowledge methods only.

5 State Information Represented as Dynamic Knowledge

State variables have their scope confined to the procedure methods. Therefore, only a procedure method can directly access a state variable. As mentioned above, a knowledge method that needs to access a state variable must do so by using a self communication to invoke a procedure method that returns its value. Alternatively, state information can be represented as dynamic knowledge clauses which can be directly accessed from other knowledge methods. The can be manipulated by the procedural methods of the object using special knowledge manipulation primitives. The can be accessed from a procedural method of an object using a query to `self`.

To illustrate how to use dynamic knowledge to represent state information, consider Program 5.1. In the **undergraduate** class, the courses a student is currently taking will be recorded as a sequence of facts

```
course_taking(Id1).
course_taking(Id2).
...
course_taking(Idk)}.
```

There is a dynamic declaration

```
    dynamic course_taking/1.
```

for it. A student object has the *enroll_for_a_course/2* method try to enroll in a specified course in the department to which it belongs, and a successful response from the department results in a fact `course_taking(CourseId)` being asserted into its dynamic knowledge. This action is done by a **self** invocation using a system built-in special primitive *acquire_knowledge/1*. The same fact will be removed, using another special primitive *remove_knowledge/1*, from the dynamic knowledge if the final grade of the course is received by the *final_grade/2* method. The removed course is then put into a list recording the courses having been taken by the student.

```
class undergraduate isa student with
{
 instance_definition
   dynamic  course_taking/1.

   states
     Dept, year := 1, Subject,
     CoursesTaken := [].

   methods
     enroll_for_a_course(CourseId, Ans) ->
       enroll_a_course(CourseId,
                          Reply) => Dept,
       if Reply == yes then
            Ans = yes,
            self ? acquire_knowledge(
               course_taking(CourseId))
       else Ans = no.

     final_grade(CourseId, Grade) ->
       self ? remove_knowledge(
                course_taking(CourseId) ),
       CoursesTaken :=
              [(CourseId,Grade)|CourseTaken].
     .
     . %other procedure methods
     . %other knowledge methods
     .
   code_definition
     .  %Parlog and Prolog programs
     .  %private to the class
     .
}.
```

Program 5.1 Using dynamic knowledge to record state information

5.1 Consistency of Access to Dynamic Knowledge State

When using state variables to record state information there is a guarantee of consistent access and update of the state. A procedure method executed on receipt of a message M sees the set of values for the state variables that result from the execution of *all* and *only* the procedure method messages that have reached the object before it, even though it can start executing before these other methods have terminated.

Because knowledge methods of an object can also execute concurrently, a similar guarantee with respect to the dynamic knowledge of the object must also be provided. In other words, the system has to promise that a knowledge method, executed on receipt of some message M', sees the dynamic knowledge that results from *all* and *only* the knowledge update messages whose execution was started before the receipt of the message M'. DK_Parlog^{++} guarantees this by handling resolutions of the knowledge update messages in a special way. A knowledge update action is only taken by an object if all previously invoked knowledge methods have completely terminated. (This means that all the needed different solutions to previous queries have been found.) In addition, no new message invoking a knowledge method will be accepted by the object until the knowledge update method has terminated.

This consistency of access to dynamic knowledge state guaranteed by DK_Parlog^{++} is stronger than that provided by the DLP language [Eli92], another distributed logic programming language in which all the methods are what we call knowledge methods. DLP only delays a state update until the first solution to each currently executing method has been found. So, on backtracking to find the next solution, the method may see a dynamic knowledge state different to that seen when the previous solution was found.

6 Related Work

Orient84/K ≡ OOP + LP + Concurrency Orient84/K was designed for the primary purpose of describing large knowledge systems, which are in turn composed of many co-operating knowledge sub-systems. It is based on a semantics called Distributed Knowledge Object Modeling(DKOM) [TI85], a knowledge system modeling mechanism using object oriented methodologies. The language combines Object Oriented Programming(OOP), Logic based Programming(LP), Demon Oriented and Concurrent Programming paradigms. Parallelism in Orient84/K is achieved by supporting distributed computation and by forking concurrent active objects which execute sequentially. No intra-method concurrency is provided.

Orient84/K is perhaps the closest in design goals to DK_Parlog^{++}. The procedural methods of its object are implemented in a distributed version of SmallTalk, and the knowledge method is implemented in Prolog. But the SmallTalk procedural methods are sequential, and the fit between SmallTalk and Prolog is much less homogeneous than that between Parlog and Prolog. For example, as we understand it, the knowledge component of another object cannot be directly queried from a Smalltalk method, only the objects local knowledge can be accessed.

DLP ≡ LP + OOP + || Distributed Logic Programming(DLP)[Eli92] was also developed with the intention to amalgamate the Logic Programming, the Object Oriented Programming and the Distributed Computation paradigms. It is an extension of Prolog with object declarations and statements for dynamic creation of objects, communication between objects and the assignment of values to state variables of objects. DLP uses a computation model which combines the Prolog computation model and a parallel object oriented model similar to that of POOL[Ame87]. Basically, an application is a collection of **object** declarations. Object instances can be created dynamically and accessed concurrently. As in DK_Parlog^{++}, public names are used for object declarations in DLP.

DLP also provides distributed computation facilities and allows concurrent execution of methods in an object. However, the method can only be what we call knowledge methods–or Prolog rules. Despite also having a distributed backtracking ability, DLP has a weaker constraint regarding when dynamic information can be changed. A state update is allowed when all current queries have returned a first solution. But in DK_Parlog^{++}, all needed solutions have to have been computed before the update is allowed. Also, parallelism in an object is only between alternative methods. Since methods are just Prolog rules, there is no concurrency within a method in DLP.

7 Concluding remarks

We have introduced most of the key features of DK_Parlog^{++}. One feature we have not mentioned is modelled on the 'become' of actors [AH87]. Instead of a state update, as a last action an instance object method can metamorphise the object into an instance of another class *without* changing the object's identity. This can be used to restrict, or totally change the methods of an instance object, when it enters a certain state.

The language is fully implemented and running on a network of machines at Imperial College. We believe that it has a unique blend of features and is a very powerful, and quite declarative, distributed object oriented programming language. Influences on its design were the previous OO extensions of CLP mentioned in the introduction, actors, and ABCL/1 [SY87] and the above mentioned DLP and Orient84/K. ABCL/1 is an actor derived language.

We are now considering restricting the use of reply variables in messages so that DK_Parlog^{++} can be implemented without the need for an underlying distributed unification scheme. The key idea here is to declare which occurrences of variables in both a sent message and a message receive pattern are reply variables, and to insist that each reply variable of a method appears in exactly one action of the method, an action which assigns a value to the variable using a special reply variable assignment operation.

Now, when a message is sent, we know what its reply variables are and we can create temporary mailboxes on the sender's machine to hold the reply bindings for these variables *if* the target object is on another machine. For each such reply variable X we also fork a process that reads the mailbox value and assigns it to X. This process suspends until the mailbox gets a value, and kills the mailbox when the value is read. The mailbox identities replace the variables in the sent message.

When a receiving method wants to 'assign' to one of its reply variables, and this assignment action finds that the variable already has a value which is a mailbox identity, it instead sends the reply value to the mailbox.

Even with this restriction on the reply variable mechanism, all the example programs of this paper can be easily rewritten and will still execute as described.

A major application area that we are investigating is Enterprise Intergration [CSW95].

References

[AH87] G. Agha and C. Hewitt. Concurrent programming using actors. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming*. MIT Press, 1987.

[Ame87] P. America, "POOL-T : A Parallel Object Oriented Language", In Yonezawa, A. and Tokoro, M., editors, *Object-Oriented Concurrent Programming*, The MIT Press, 1987.

[CC93] D. Chu, K. L. Clark, "IC-PrologII: a Multi-threaded Prolog System", In *Proceedings of the ICLP'93 Post Conference Workshop on Concurrent, Distributed & Parallel Implementations of Logic Programming Systems*, Budapest, 1993.

[CC93] Y. Cosmadopoulos and D. A. Chu, *IC-Prolog II Reference Manual*, Logic Programming Section, Dept. of Computing, Imperial College, London, 1993.

[CDB+93] J. Crammond, A. Davison, A. Burt, M. Huntbach, M. Lamand, Y. Cosmadopoulos, and D. Chu. The parallel parlog user manual. Dept. of Computing, Imperial College, 1993.

[CG86] K. L. Clark and S. Gregory. Parlog : Parallel programming in logic. *ACM transactions on Programming Languages and Systems*, 8(1):1–49, 1986.

[CW94] K. L. Clark and T. I Wang. Distributed object oriented logic programming. ICOT Fifth Generation Computer System Workshop on Heterogeneous Cooperative Knowledge-Bases, 1994.

[CSW95] K. L. Clark and N. Skarmeas and T. I Wang. Distributed object oriented logic programming as a tool for enterprise modelling. *Proceedings of IFIP TC5 Working Conference on Modelling and Methodologies for Enterprise Integration* To be published by Chapman and Hall

[CW96] K. L. Clark and T. I Wang., "D_Parlog^{++} - Object Oriented Logic Programming with Distributed Active Classes", in Goldsack S.J. and Kent S.J.H., editors, *Formal Methods and Object Technology*, Springer Verlag, to appear (1996).

[Dav89] A. Davison. *Polka:A Parlog Object Oriented Language*. PhD thesis, Imperial College, 1989.

[Eli92] A. Eliëns, *DLP A Language For Distributed Logic programming-Design, Semantics and Implementation*, John Wiley & Sons, England, 1992

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[KTMB87] Kenneth M. Kahn, Eric D. Tribble, Mark S. Miller, and Daniel G. Bobrow. Vulcan: Logical concurrent objects. In P. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

[McC92] F. G. McCabe, *L&O: Logic and Objects.* International series in Computer Science. Prentice-Hall International, 1992.

[ST83] E. Y. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog. In *New Generation Computing.* 1983.

[SY87] E. Shibayama and A. Yonezawa. Distributed computing in abcl/1. In A. Yonezawa and M. Tokoro, editors, *Object Oriented Concurrent Programming.* MIT Press, 1987.

[TI85] M. Tokoro, and Y. Ishikawa, "Orient84/K : A Language with Multiple Paradigms in the Object Framework" , in *Proceeding of the 19th Hawaii International Conference on System Sciences*, Honolulu, January 1986.

[Wan95] T. I Wang. *Distributed Object-oriented Logic Programming.* PhD thesis, Imperial College, 1995.

[YC88] K. Yoshida and T. Chikayama. A'um-a stream-based concurrent object-oriented language. In *Proceeding of FGCS'88.* ICOT, 1988.

NOTES